# NAVAL POSTGRADUATE SCHOOL
## Monterey, California



# THESIS

**TESTING OF A READ PREDICTION BUFFER INTEGRATED CIRCUIT AND DESIGN OF A PREDICTIVE READ CACHE**

by

Max E. Aguilar F.

March 1995

Thesis Co-Advisors: Douglas J. Fouts
Timothy Shimeall

Approved for public release; distribution is unlimited.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY (Leave Blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | March 1995 | Master's Thesis |

**4. TITLE AND SUBTITLE**

TESTING OF A READ PREDICTION BUFFER INTEGRATED CIRCUIT AND DESIGN OF A PREDICTIVE READ CACHE

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**

AGUILAR, Max Enrique

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Naval Postgraduate School
Monterey, CA 93943-5000

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSORING/ MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**

Approved for public release; distribution is unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

The objective of this research work was to evaluate and test the Read Prediction Buffer integrated circuit (IC). This IC attempts to decrease main-memory latency by predicting the next data cache read miss address and pre-fetching the data before the miss actually occurs in the cache. The motivation for its testing is that, if correct, the chip will significantly improve the speed of imbedded microprocessors which are so prevalent in modern equipment.

The approach taken, was to place the RPB between a Pattern Generator Module and a State- Timing logic Analysis Module. The pattern generator was programmed to generate test cases. The output signals of this module were applied to the input pins of the chip. The chip's response was then captured and analyzed using the logic analysis module. Results showed that the chip worked correctly and fully implemented the intended algorithm. However, an evaluation of its architecture indicated two major problems; a) The RPB provides an additional level of latency to the memory structure when a predicted

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES | 16. PRICE CODE |
|---|---|---|
| VLSI (very large scale integration) design; memory address prediction; VERILOG; EPOCH; CMOS; cache performance improvement. | 154 | |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

13. Abstract (continue)

address is in error, b) Every time there is a displacement change (which occurs at branches) the RPB predicted address will be in error.

These two factors forced the redesign of the RPB, giving birth to the Predictive Read Cache. In the PRC, the first problem was solved by reallocating the chip's position in the memory hierarchy. The IC was converted from a memory controller device to a snooping device. The second problem was eliminated by increasing the number of predictive lines from 1 to 128. This means that the PRC is now able to track 128 different displacements.

Approved for public release; distribution is unlimited

# TESTING OF A READ PREDICTION BUFFER INTEGRATED CIRCUIT AND DESIGN OF A PREDICTIVE READ CACHE

by


Max E. Aguilar F.
Lieutenant J.G, Honduran Navy
B.S., US Naval Academy, 1988


Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1995

Author: ______________________
Max E. Aguilar F.

Approved By: ______________________
Douglas Fouts, Thesis Co-Advisor

______________________
Timothy Shimeall, Thesis Co-Advisor

______________________
Ted Lewis, Chairman
Department of Computer Science

______________________
Michael A. Morgan, Chairman
Department of Electrical And Computer Engineering

# ABSTRACT

The objective of this research work was to evaluate and test the Read Prediction Buffer integrated circuit (IC). This IC attempts to decrease main-memory latency by predicting the next data cache read miss address and pre-fetching the data before the miss actually occurs in the cache. The motivation for its testing is that, if correct, the chip will significantly improve the speed of imbedded microprocessors which are so prevalent in modern equipment.

The approach taken, was to place the RPB between a Pattern Generator Module and a State-Timing logic Analysis Module. The pattern generator was programmed to generate test cases. The output signals of this module were applied to the input pins of the chip. The chip's response was then captured and analyzed using the logic analysis module. Results showed that the chip worked correctly and fully implemented the intended algorithm. However, an evaluation of its architecture indicated two major problems; a) The RPB provides an additional level of latency to the memory structure when a predicted address is in error, b) Every time there is a displacement change (which occurs at branches) the RPB predicted address will be in error.

These two factors forced the redesign of the RPB, giving birth to the Predictive Read Cache. In the PRC, the first problem was solved by reallocating the chip's position in the memory hierarchy. The IC was converted from a memory controller device to a snooping device. The second problem was eliminated by increasing the number of predictive lines from 1 to 128. This means that the PRC is now able to track 128 different displacements.

## TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# I. INTRODUCTION

## A. THEORY OF OPERATION

The Read Prediction Buffer integrated circuit (IC) was designed and implemented by Gary J. Nowicki as part of his thesis work in 1992 [Ref. 1]. The RPB is basically a buffer, attached to individual memory modules, that stores one word of data corresponding to a predicted memory address. When a read miss occurs in the on-chip data cache, the RPB IC compares the miss address to the predicted one. If both addresses are equal, the buffer transfers its pre-fetched data to the on-chip data cache, reducing the apparent latency of the main memory access. Otherwise, it allows the read operation to proceed normally. If a write access occurs, the RPB performs, when necessary, a write through operation to maintain data coherency.

The chip utilizes a displacement-based prediction algorithm in an attempt to predict the next data cache read miss address. The algorithm is simple enough to be implemented inexpensively in dedicated hardware and it has proved to be amazingly successful for programs with strong spatial and temporal locality[Ref. 2]. Figure 1 describes this algorithm in detail. When an address arrives, its value is compared to the predicted one . This operation determines whether the address conforms or not. Conforming means the triggering of a pre-determined sequence of operations that allows the completion of some action previously started based on the predicted value. Regardless of the boolean outcome, the algorithm proceeds to calculate the next predicted value. This is done by extracting the offset between the current address and the previous address and then adding that offset to the current address. To clarify this, the equation and associated simplification is presented in Figure 2. In the Read Prediction Buffer chip, conforming is the transfer of the pre-fetched data to the on-chip data cache and the action based on prediction is the fetch of data stored in memory at the predicted address.

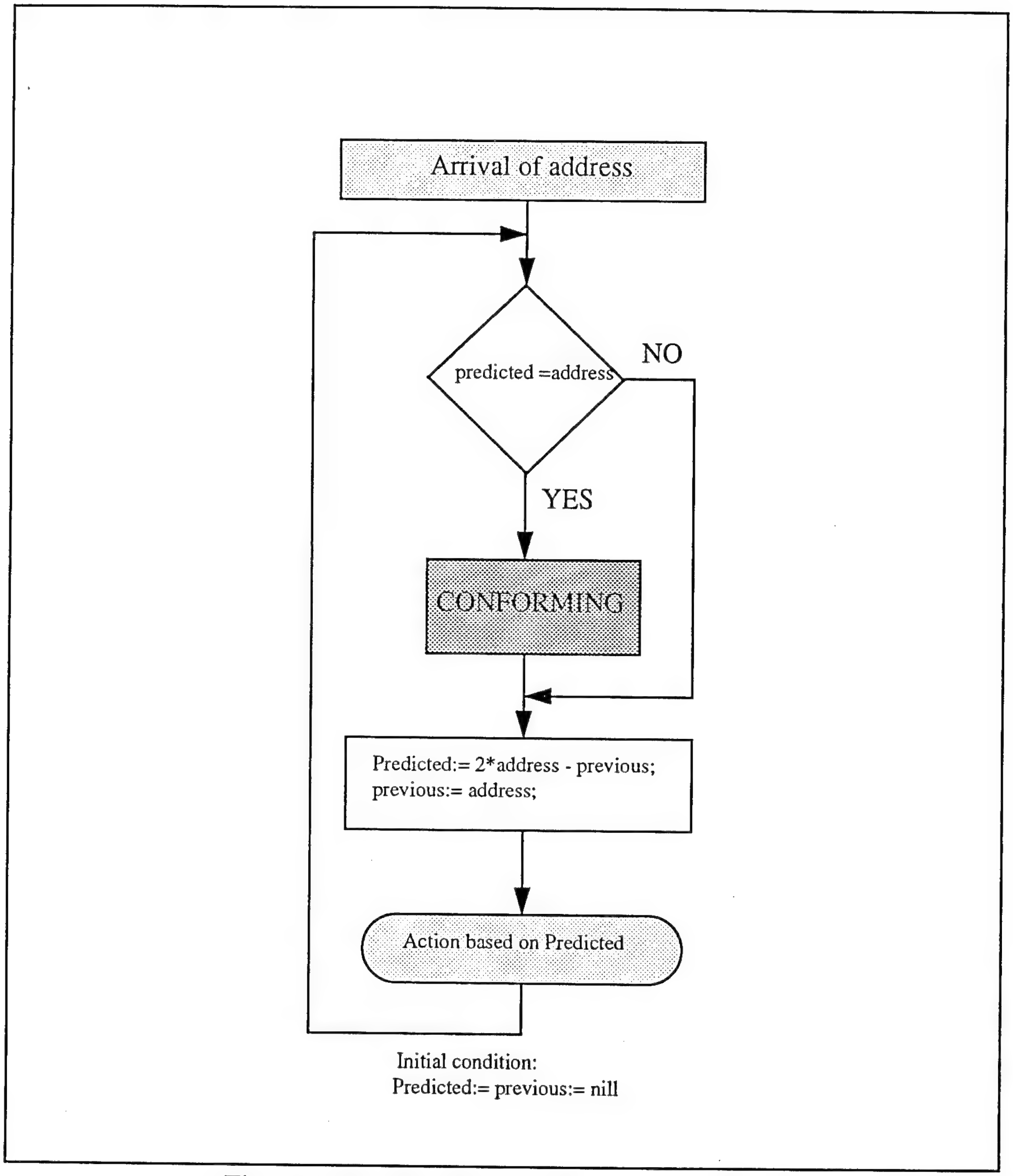



**Figure 1: The Displacement-Based Algorithm**

1. OFFSET = $C_A - P_A$

2. Predicted = $C_A$ + OFFSET

Substituting 1 into 2;

3. Predicted = $C_A + (C_A - P_A)$

Simplifying;

4. Predicted = $2*C_A - P_A$

$C_A$: Current address

$P_A$: Previous address

**Figure 2: Predictive Equation**

## B. ENHANCEMENTS

As explained in the previous section, the RPB was conceived to improve memory performance. However, a closer analysis of its architecture indicates two major drawbacks:

- System Performance Degradation: The buffer provides an additional level of latency to the memory structure when a predicted address is in error.
- Branch Sensitive: Every time there is a displacement change (which occurs at branches) the predicted address will be in error.

The solution to these problems forced considerable changes in the architecture of the buffer. In fact, its placement in the memory hierarchy has completely changed. This new design has given birth to the "Predictive Read Cache" chip which could become an alternative to on-chip second level cache memories [Ref. 2].

In the PRC, the first problem was eliminated by reallocating the chip. It now conforms to a different system configuration as shown in Figure 3. The chip no longer forms part of the main-memory module. Instead, it acts as a stand-alone device that snoops read/write operations between the on-chip data cache and main-memory. This way, the device won't

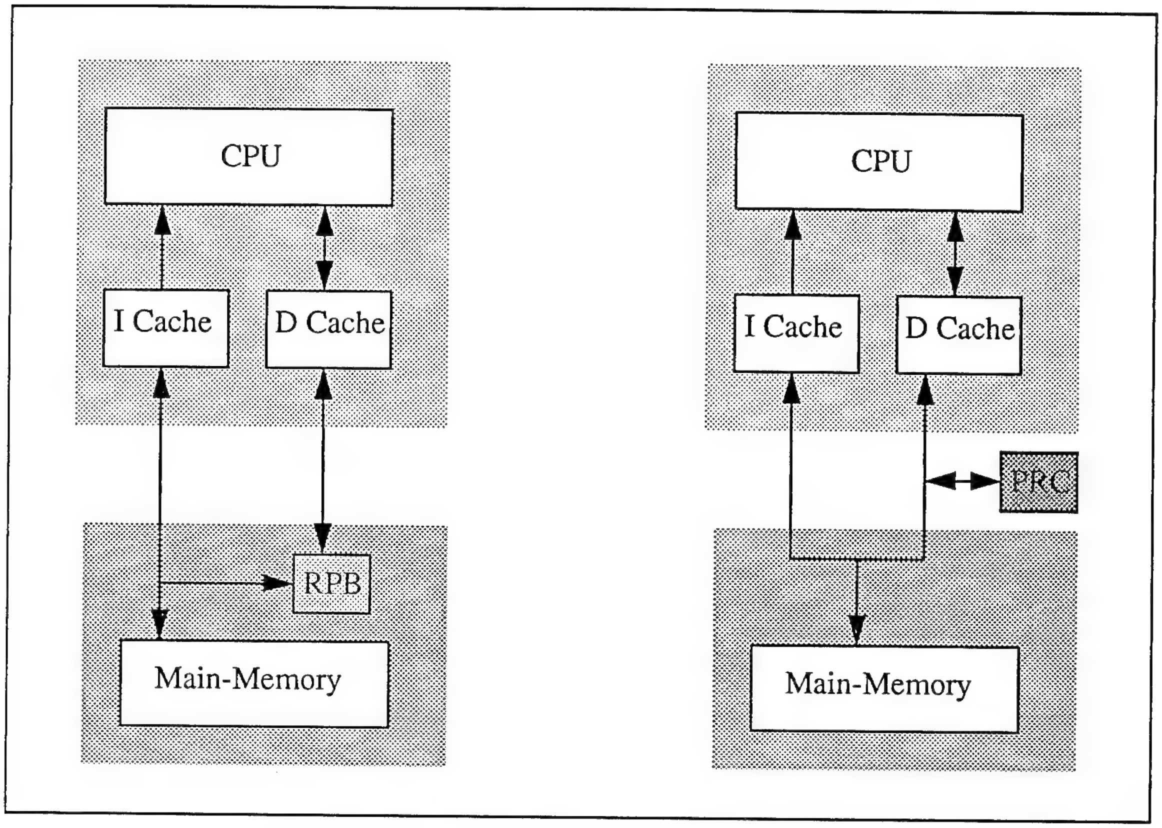



**Figure 3: RPB and PRC System Location**

interfere with the read operation if it does not correctly predict the read miss address. Otherwise, it stops the read operation and provides its pre-fetched data to the data cache. This process cuts the normal read time almost in half.

This new configuration requires an interface with appropriate handshake signals to interact with the cpu and main memory modules. The interface chosen was that of the Motorola-IBM Power PC603 chip.

The second problem was eliminated by increasing the number of predictive lines from 1 to 128. This means that the chip is now able to track 128 different displacements. This arrangement not only decreases the branch sensitivity but also increases the probability of a hit.

## C. RESEARCH GOALS

It is the goal of this Thesis to fully document the RPB testing and find out the following:

- Does the chip work at all?
- Does it correctly implement the intended algorithm?
- What is the minimum and maximum failure rate?
- What sections of the chip do or do not work?
- Is there a latchup problem?
- What kind of noise margins does it have?
- What is the power consumption?

The motivation for such testing is that the RPB is the first IC designed and implemented at the Naval Postgraduate that actually has over 10,000 transistor in it (Previous projects were of smaller complexity). Therefore, it is of significant importance to the NPS VLSI instructors to fully test the chip in order to evaluate the used CAD tools, taught techniques, and available library. Furthermore, the process of testing an IC with a nontrivial algorithm is not well explored in the literature. This work may serve as an example and could be used as a guide to future IC testing.

An additional goal is to proceed with the design and implementation of the PRC. The complexity of the PRC far exceeds that of the RPB and its completion may not be possible as part of this work. However, whatever advances are made, they will be of great significance to future researches.

## D. REQUIRED EQUIPMENT AND CAD TOOLS

The main equipment utilized for the testing of the RPB chip was the Naval Postgraduate School's HP16500B Logic Analysis System. The design and layout of the PRC was done on the NPS' Sun SPARCstations utilizing Mentor Graphics' Design Architect, Cadence's Verilog-XL and Cascade's Epoch 3.1.

### 1. Hewlett-Packard Logic Analysis System

The HP16500B is the mainframe of the Hewlett-Packard Logic Analysis System. It uses the Motorola 68EC030 microprocessor and is equipped with an 85 Mbyte DOS-formatted high-reliability hard drive and a single 3.5-inch 1.44 floppy disk drive.The system offers a modular structure for plug-in cards [Ref. 3]. There are many test and measurement modules available that fit into the mainframe. The following briefly describes the add-on modules acquired by the Naval Postgraduate School.

#### *a. Acquisition Board*

The NPS purchased one HP16550A 100-Mhz State/ 500 Mhz Timing Logic Analyzer board. The module has 96 data channels and six clock/data channels. Captured data can be display as data listings or waveforms, and can be plotted on a chart or compared to a reference image.[Ref. 4]

#### *b. Stimulus Boards*

The NPS also purchased one HP16520A Pattern Generator Master Card and two HP16521A Pattern Generator Extension Cards. The PGM is a general purpose digital stimulus. The master card offers the minimum configuration of 12 data and 3 strobe channels, a clock out channel, and three dedicated input qualifier channels. Each expansion card offers expandability with 48 additional data out channels (up to four expansion cards can be connected to a single master card). The module exhibits a 4 K bit per channel memory depth, TTL and ECL interface levels, intermodule triggering, and clock rates between 5 Khz and 50 Mhz.[Ref. 5]

### 2. Sun SPARCstations

The SPARCstation 10 is a uniprocessor desktop computer, manufactured by Sun Microsystems Computer Corporation. The workstation utilizes the SuperSPARC microprocessor which can execute three instructions concurrently, enabling superior integer and floating point performance for complex computation. The Naval Postgraduate

School has a variety of models configured differently, but in general with a 64 MB of RAM memory and two 424MB internal hard drive, which are mainly used for swap space since the system mounts several large files systems from remote servers.[Ref. 6]

### 3. Mentor Graphics' Design Architect

Design Architect is a multi-level design environment in which top-down designs are captured at the architectural, logic and circuit levels. The environment includes editors for Schematic, Symbols, Logical Cable, and VHDL designs. Compiler for VHDL also included.[Ref. 7]

### 4. Cadence's Verilog-XL

Verilog is a hardware descriptive language (HDL) and simulator specification capable of describing circuits structurally, functionally or a combination of both.[Ref. 8]

### 5. Cascade's Epoch 3.1

"Epoch is a complete physical IC design system utilizing state-of-the-art double and triple metal technology with finite state machine synthesis and automatic placement, routing and buffer/power rail sizing. Epoch accepts input from Verilog, VHDL, Valid GED™, Synopsys®, VIEWlogic®, Mentor Graphics Design Architect®, Cadence Composer™ and EDIF. It provides layout-based simulation output in Verilog, VHDL, QuicksimII™ and EDIF formats and outputs geometry in GDSII and CIF mask generator formats."[Ref. 9]

## E. THESIS STRUCTURE

The implementation of the algorithm and the testing procedures for the Read Predictive Buffer chip are discussed in Chapter II. Fundamental block designs of the Predictive Read Cache are explained in Chapter III. Implementation details of different PRC components are shown in Chapter IV. Simulation of different components are presented in Chapter V. Chapter VI gives the thesis conclusions and further recommendations.

# II. TESTING OF THE READ PREDICTIVE BUFFER CHIP

## A. IMPLEMENTATION OF THE ALGORITHM

The RPB makes use of a displacement-based algorithm which extracts the offset between two consecutive read accesses and adds the offset to the most recent read access. The implementation of the algorithm in the RPB, shown in Figure 4, is straight forward. It utilizes two register files to hold the current and previous addresses, an adder and subtracter to predict the next data read address[1], and a comparator to perform the boolean operation on the requested and predicted addresses.

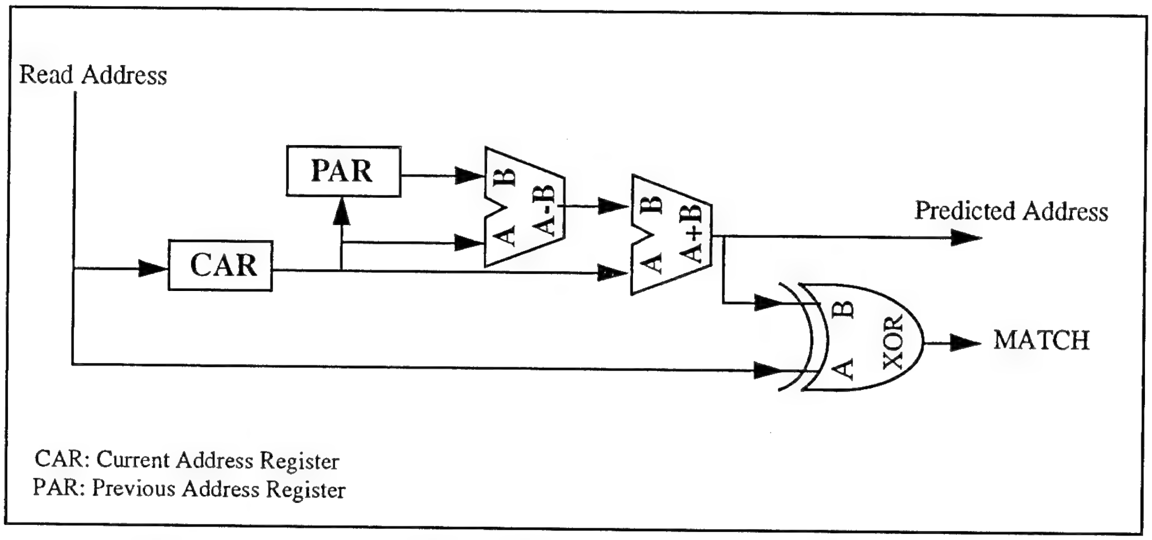



**Figure 4: Implementation of The Algorithm in the RPB. After Ref [1]**

The control sequence performed on the datapath of Figure 4 is explained in Figure 5. Notice that this particular implementation does not compare values until the third Read access occurs. Furthermore, it uses an extra register, namely the CAR, to hold the value of the current read access. Figure 6 presents the finite state machine responsible for this

1. Notice from Figure 4, that RPB implements equation 3 on Figure 2.

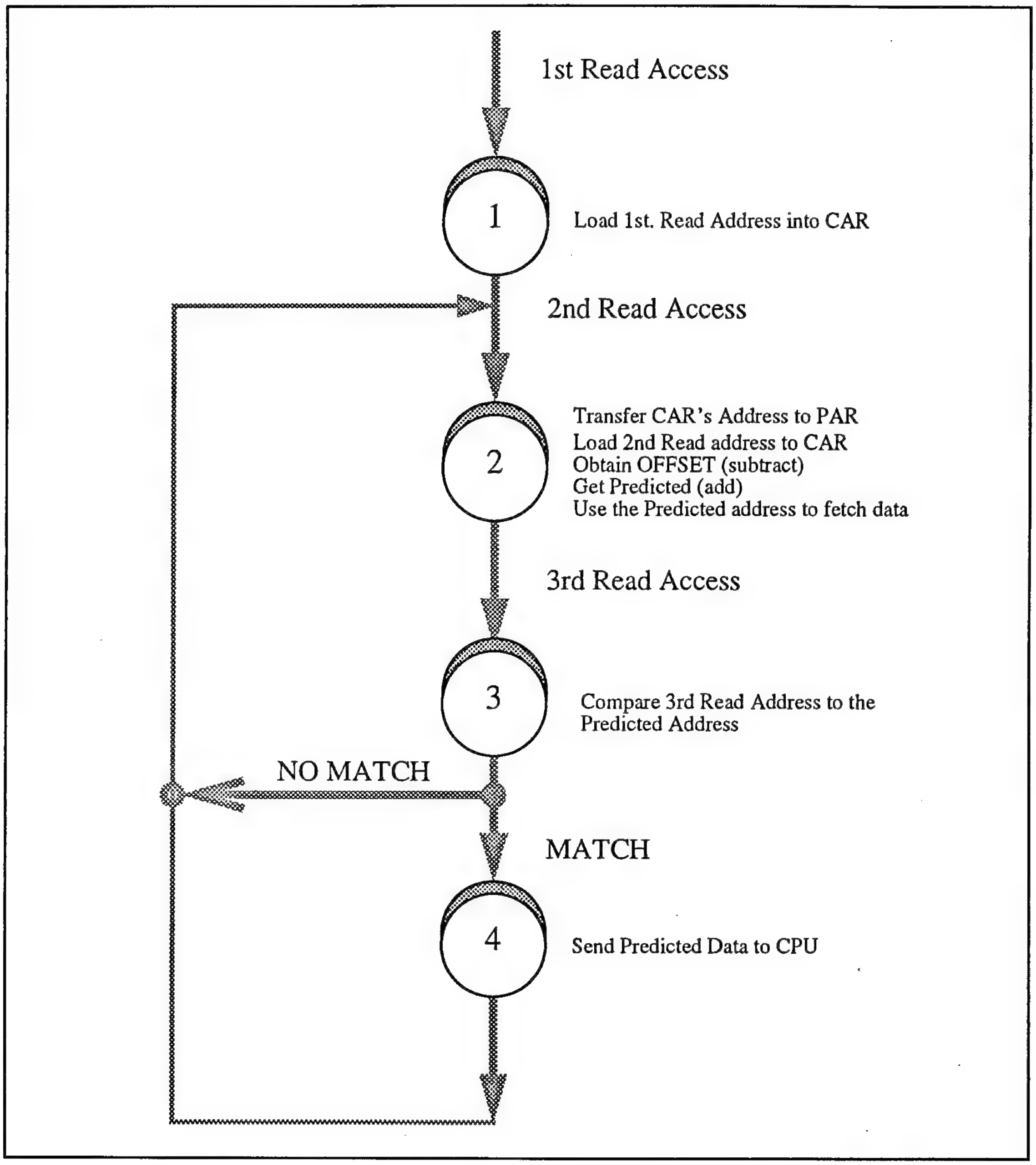



**Figure 5: Basic RPB Algorithm Flow Chart. From Ref [1]**

sequence. The machine generates the necessary internal signals to control the flow. In addition, it sends and receives external signals to interact with the memory module. This state diagram can easily be understood without going into details. The states on the right hand side are used to count the number of read accesses received, and perform the

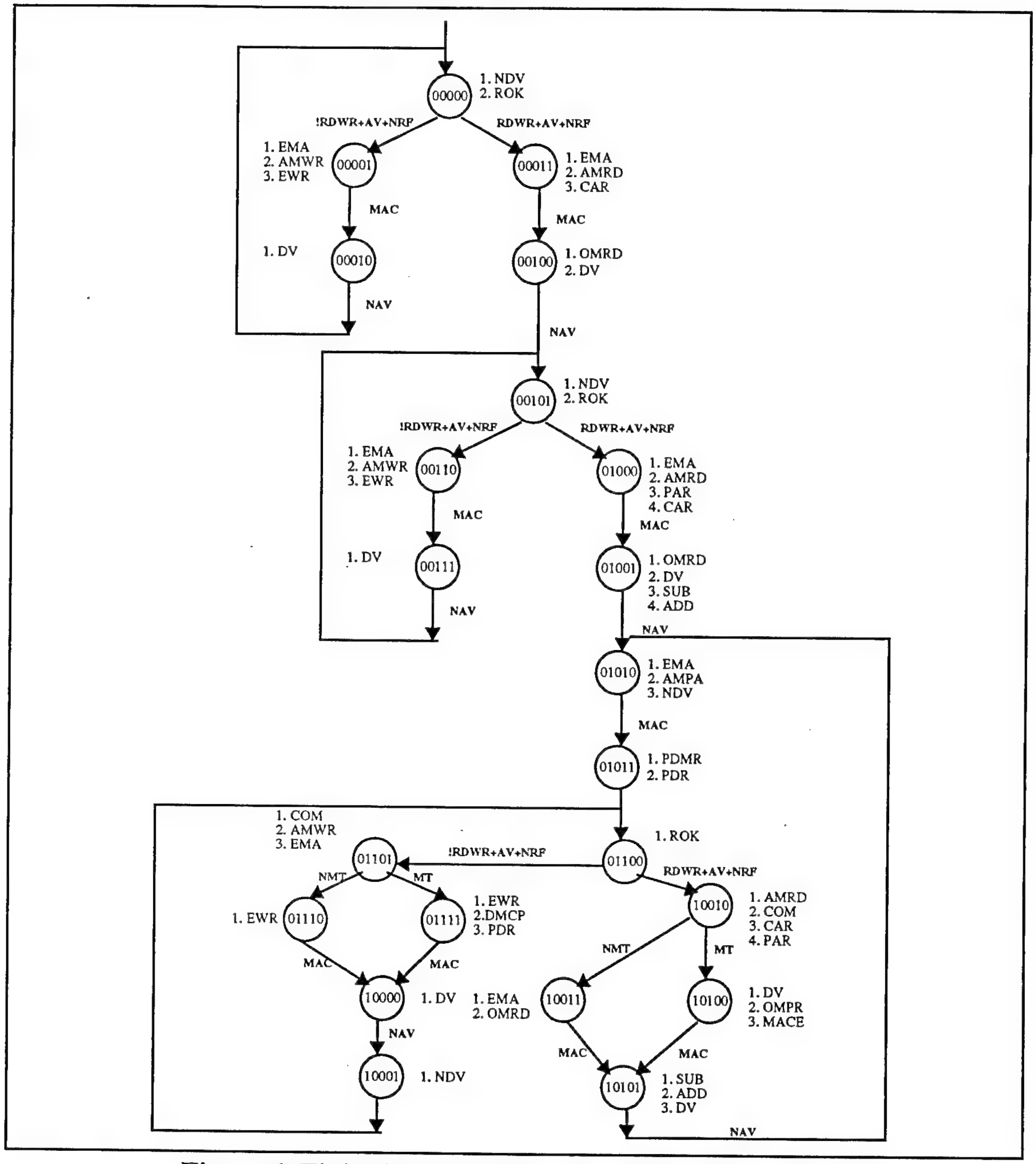



**Figure 6: Finite State Machine Flow Chart. From Ref [1]**

prediction and comparison sequence presented in Figure 5. The states forming a loop on the left side (1,2,5,6,7,12-17) are used to execute a write-through operation whenever a write access is received. Verifying proper functionality of this state machine constitutes the core of the testing. The following section describes the procedures and results for such testing.

## B. PROCEDURE

Testing was divided into two phases. The Visual and the Functional phase. In the visual one, the magic files (layouts), extractions, and spice simulations were inspected. A great deal of time was spent looking for obvious errors and design rule violations. Negative results were obtained, therefore, no documentation is provided. For a review of related information, the reader is referred to the original document prepared by Nowicki in 1992, [Ref. 1].

In the functional phase, the chip was treated as a black box. Signals from the HP16520A Pattern Generator were applied to the input and control pins of the chip. The chip's response was then captured and analyzed using the HP16550A State/Timing logic analysis module. Acquisition of expected responses ensured the proper functionality of the chip. The rest of this chapter is dedicated to presenting detailed information concerning the setting of the test bench, generated stimulus, acquisition and analysis of responses, and other tests performed.

### 1. Test Bench Setting

Figure 7 presents the basic idea of the test bench used. Board1 and Board3 are located within the HP16500B Logic Analysis System. Board2 was designed and built to hold the RPB chip. The board was constructed utilizing the information presented in Appendix D.

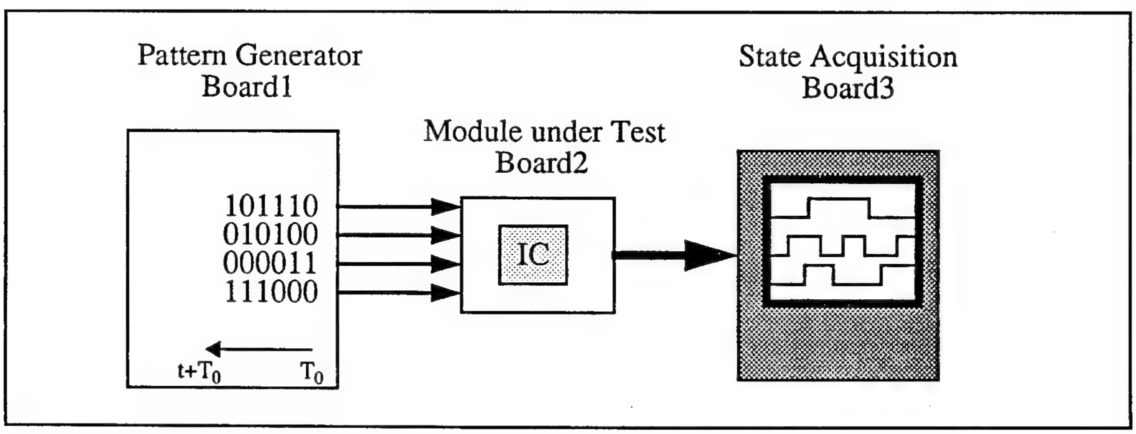



**Figure 7: Test Bench Block Diagram**

## 2. Generated Stimulus

The HP pattern generator module was programmed to generate test cases for the finite state machine. Although an exhaustive set of cases could have been generated, the process of analyzing every single response would have consumed an extraordinary amount of time. Besides, the purpose was to find out functionality, not reliability. The idea then was to create programs that would force the state machine to enter all of its states. One such program is shown in Figure 8.

A detail explanation of what the program does is given in the next section where analysis of the chip's response is performed. For now, it suffices to say that the program consist of a body plus the three macros shown in Figure 9. These macros constitute the basic building blocks. The first macro, named **write**, enables the proper external signals in order to simulate a write access. Likewise, the **read** macro (second one) simulates a read access. Notice the only difference between these two macros is the level of the *RDWR* signal and the channel in which data is sent. Read data is sent through channel *H* to simulate memory fetching. On the other hand, write data is sent through channel *G* to simulate cpu writes. The last macro, MAC, enables the *MAC* signal.

## 3. Data Acquisition and Analysis

This section explains in detail the generated stimulus and the chip's response to those stimulus. Furthermore, it points out the different sections of the state machine and datapath involved in that response. Figure 10 presents the chip's block diagram which identifies the I/O signals, internal signals, and the main components of the chip. This diagram, along with the state machine diagram previously presented, will be used to trace the working sections of the chip.

PATTERN GENERATOR PROGRAM LISTING

| Instr | ADDR | NAV | AV | NRF | MAC | RDWR | H | G | CLKA | CLKB |
|---|---|---|---|---|---|---|---|---|---|---|
| | HEX | BIN | BIN | BIN | BIN | BIN | HEX | HEX | BIN | BIN |
| 00 SIGNAL IMB | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 01 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 02 WRITE | 000001 | “ | “ | “ | “ | “ | “““ | 001 | 1 | 1 |
| 03 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 07 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 08 READ | 000001 | “ | “ | “ | “ | “ | 001 | “““ | 1 | 1 |
| 09 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 13 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 14 WRITE | 000006 | “ | “ | “ | “ | “ | “““ | 006 | 1 | 1 |
| 15 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 19 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 20 READ | 000002 | “ | “ | “ | “ | “ | 002 | “““ | 1 | 1 |
| 21 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 25 MAC | | “ | “ | “ | “ | “ | “““ | “““ | 1 | 1 |
| 26 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 29 WRITE | 000003 | “ | “ | “ | “ | “ | “““ | 00F | 1 | 1 |
| 30 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 34 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 35 READ | 000003 | “ | “ | “ | “ | “ | 003 | “““ | 1 | 1 |
| 36 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 40 MAC | | “ | “ | “ | “ | “ | “““ | “““ | 1 | 1 |
| 41 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 44 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 45 READ | 000005 | “ | “ | “ | “ | “ | 005 | “““ | 1 | 1 |
| 46 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 50 MAC | | “ | “ | “ | “ | “ | “““ | “““ | 1 | 1 |
| 51 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 54 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 55 WRITE | 000006 | “ | “ | “ | “ | “ | “““ | 006 | 1 | 1 |
| 56 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 60 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 61 WRITE | 000007 | “ | “ | “ | “ | “ | “““ | 007 | 1 | 1 |
| 62 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 66 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |
| 67 READ | 000007 | “ | “ | “ | “ | “ | 001 | “““ | 1 | 1 |
| 68 PARAMETERS | | “ | “ | “ | “ | “ | “““ | “““ | “ | “ |
| 72 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | 000 | 000 | 1 | 1 |

**Figure 8: Pattern Generator Main Program Listing**

PATTERN GENERATOR MACRO LISTING

| Instr | ADDR | NAV | AV | NRF | MAC | RDWR | H | G | CLKA | CLKB |
|---|---|---|---|---|---|---|---|---|---|---|
| | HEX | BIN | BIN | BIN | BIN | BIN | HEX | HEX | BIN | BIN |
| 00 **WRITE** | ADDRS | P1 | P1 | P1 | P1 | P1 | P1 | GDATA | CKA | CKB |
| 01 PARAMETERS | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 |
| 02 | ADDRS | 1 | 1 | 1 | 0 | 0 | “““““ | GDATA | CKA | CKB |
| 03 REPEAT 2 | ADDRS | 1 | 0 | 0 | 0 | 0 | “““““ | GDATA | CKA | CKB |
| 04 | ADDRS | 1 | 0 | 0 | 1 | 0 | “““““ | GDATA | CKA | CKB |
| 05 WAIT 1XX | ADDRS | 0 | 0 | 0 | 0 | 0 | “““““ | GDATA | CKA | CKB |
| 06 | ADDRS | 1 | 0 | 0 | 0 | 0 | “““““ | “““““ | CKA | CKB |
| END OF MACRO | | | | | | | | | | |
| 00 **READ** | ADDRS | P1 | P1 | P1 | P1 | P1 | HDATA | P1 | CKA | CKB |
| 01 PARAMETERS | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 |
| 02 | ADDRS | 1 | 1 | 1 | 0 | 1 | HDATA | “““““ | CKA | CKB |
| 03 REPEAT 2 | ADDRS | 1 | 0 | 0 | 0 | 1 | HDATA | “““““ | CKA | CKB |
| 04 | ADDRS | 1 | 0 | 0 | 1 | 1 | HDATA | “““““ | CKA | CKB |
| 05 WAIT 1XX | ADDRS | 0 | 0 | 0 | 0 | 1 | HDATA | “““““ | CKA | CKB |
| 06 | ADDRS | 1 | 0 | 0 | 0 | 1 | “““““ | “““““ | CKA | CKB |
| END OF MACRO | | | | | | | | | | |
| 00 **MAC** | ADDRS | P1 | P1 | P1 | P1 | P1 | P1 | P1 | CKA | CKB |
| 01 PARAMETERS | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 |
| 02 REPEAT 2 | 000000 | 1 | 0 | 0 | 0 | 0 | “““““ | “““““ | CKA | CKB |
| 03 | “““““““““““ | 1 | 0 | 0 | 1 | 0 | “““““ | “““““ | CKA | CKB |
| 04 | “““““““““““ | 1 | 0 | 0 | 0 | 0 | “““““ | “““““ | CKA | CKB |
| 05 | “““““““““““ | 1 | 0 | 0 | 0 | 0 | “““““ | “““““ | CKA | CKB |
| END OF MACRO | | | | | | | | | | |

**Figure 9: Pattern Generator Macro Listing**

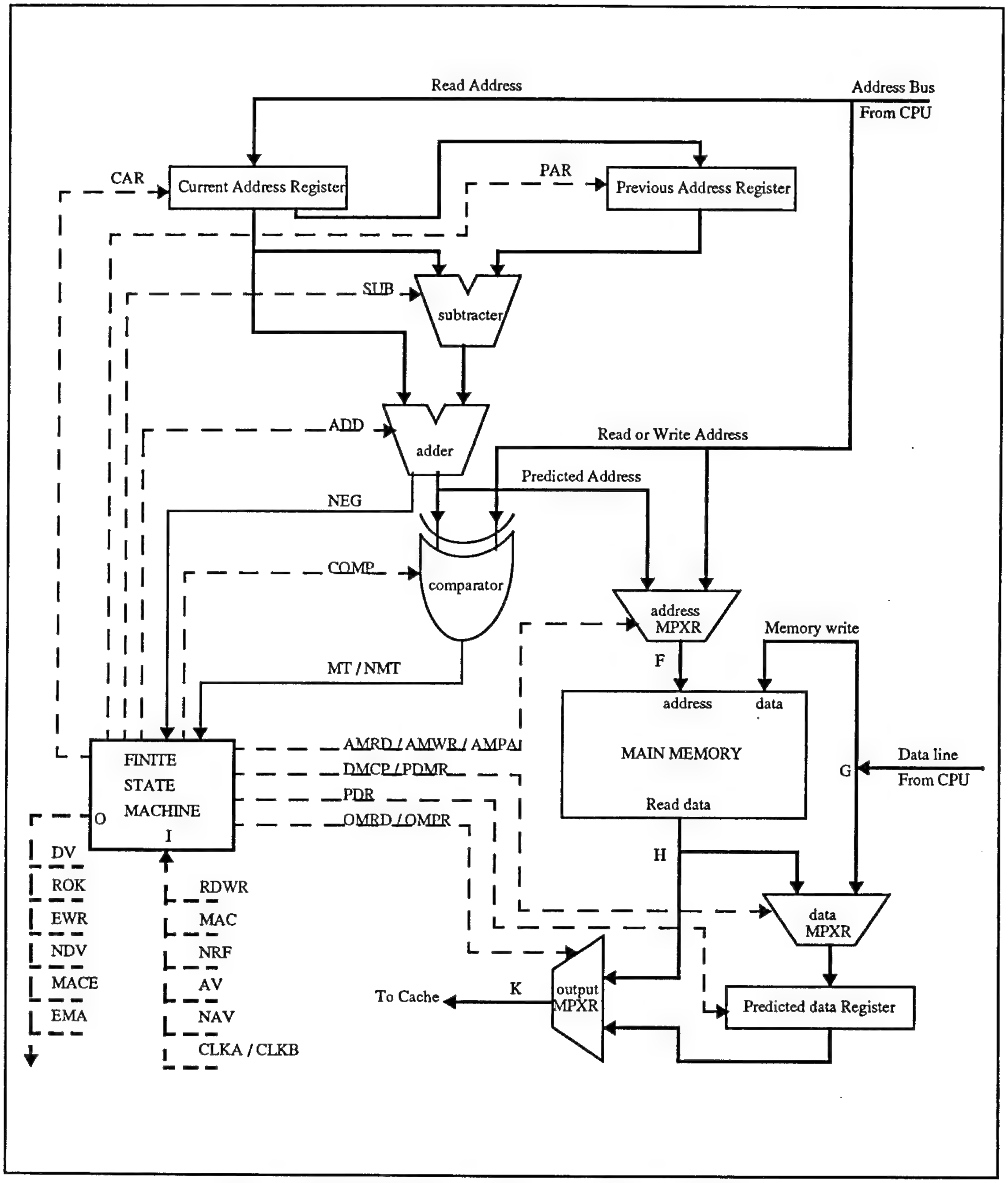



**Figure 10: RPB Block Diagram. From Ref [1]**

### *a. Overview*

The very first step was to make sure that the chip's state machine was in fact in its initial state. That was accomplished by enabling only the clock signals. This is clearly shown in Figure 11. After just one clock cycle the chip was ready for input. This is indicated by the assertion of the *NDV* and *ROK* signals (state 0d). The clocks were left to run free for an arbitrary period of time to make sure that the state was stable. At this point, the

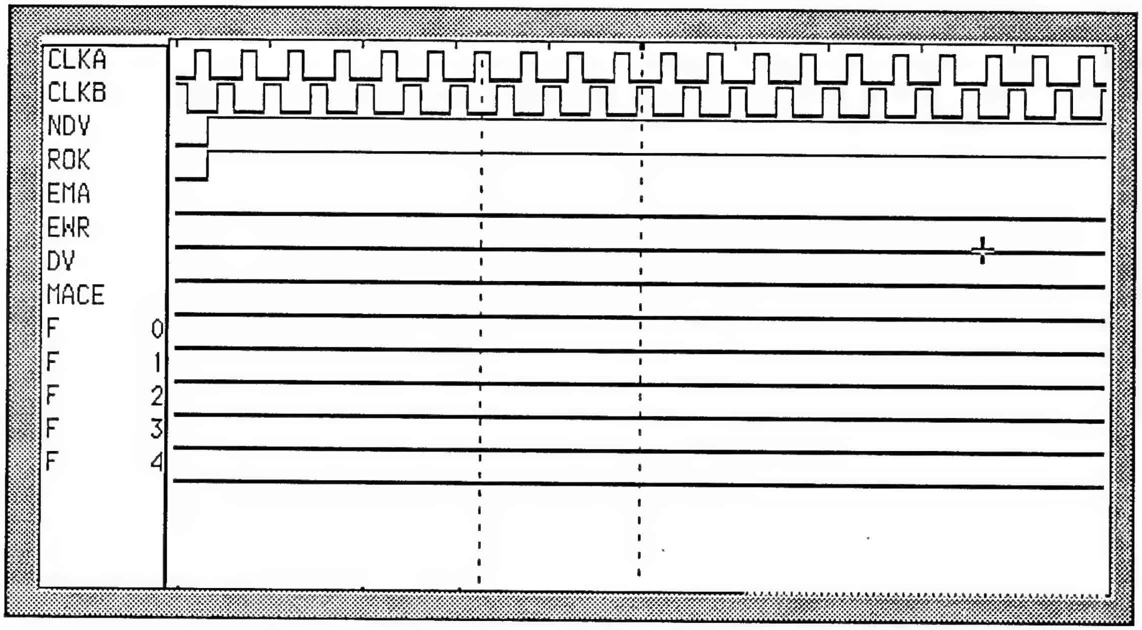



**Figure 11: Chip Response to Clock signals**

pattern generated by the program was applied to the input signals of the chip. The logic analyzer connected to the output signals of the RPB captured the response. This response is shown is Figure 12. For convenience, the figure has been partitioned into 5 sections (the division is marked by the vertical dash lines). Each section is explained and analyzed next in order to verify that every state and every component responded correctly. Although not numbered, the sections are taken sequentially from left to right. It is important to explain the convention used in this figure and following ones. The output signals are displayed above the clock line, and the inputs below it. All signals except for *NAV* are asserted high

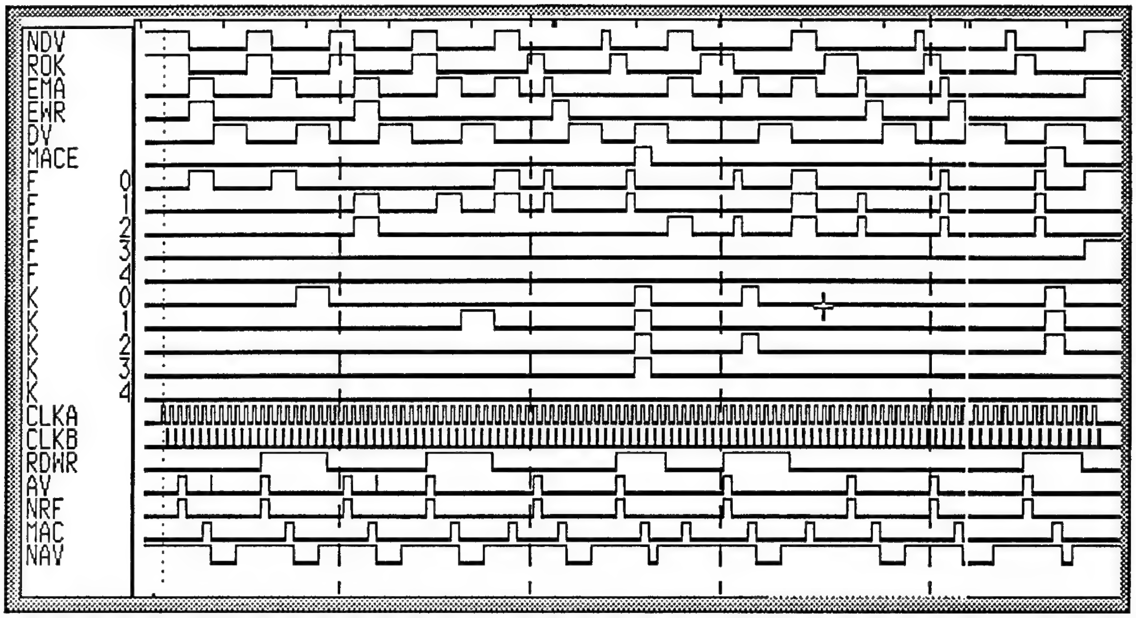



**Figure 12: Chip Response to the Generated Pattern**

and rising edge triggered. *NAV* is asserted low and trailing edge triggered. For the following figures, the number below the asserted pulse refers to the state in which that signal (and vertical ones) are asserted. The lines with arrows, indicate the inputs that force a change of state. The signal's name written below the asserted pulse is the internal signal that enables that external signal to appear. Last, the *F* signal is a 22-bit signal, and the *K* signal is an 8 bit signal. Only the first 4 bits of each signal are displayed since the testing numbers are kept small.

### b. *Section 1*

At line 00, the program sends an internal signal to synchronize the pattern generator with the timing analyzer module. Next, it simulates a write access followed by a read access (line 02 through 13). Both accesses are done to location 000001H. The written value is unimportant at this point, and the read value was chosen to be 001H. This completes the stimulus for this stage. Figure 13 presents the captured waveform for this section. Initially, the chip is in state 0 and asserts the *NDV* and *ROK* signals, indicating that it is ready for input. The RPB detects a write access whenever the *RDWR* signal is negated

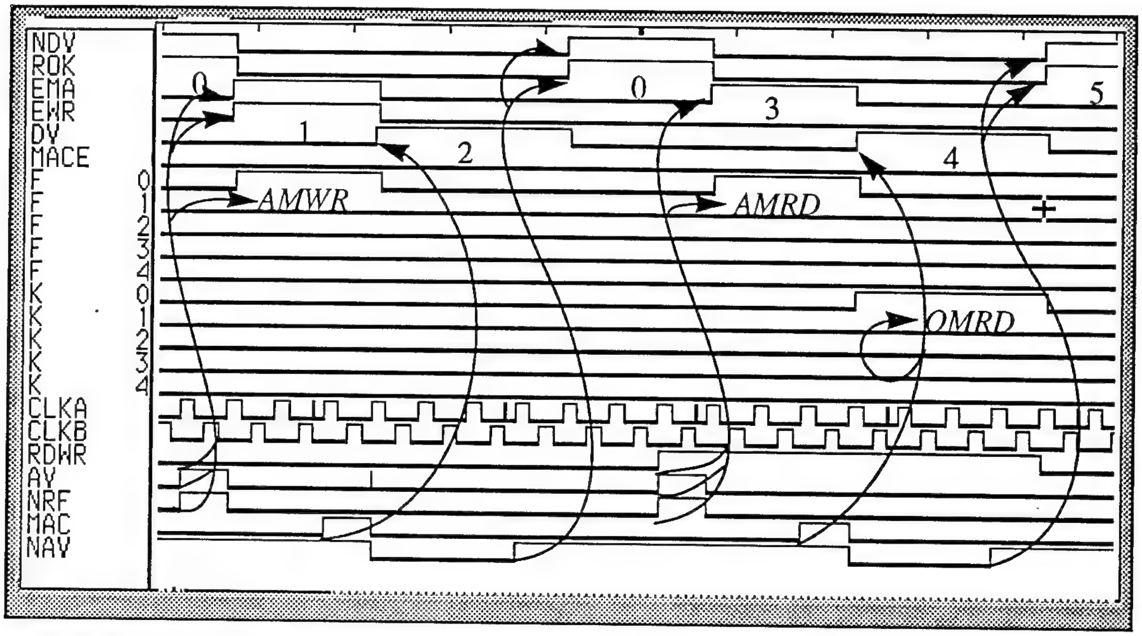



**Figure 13: Section 1, Captured Waveform**

and both *AV* and *NRF* are asserted (a read access is when all three signals are asserted). When the first write occurs, the machine enters state 1, asserting *EMA, EWR,* and *AMWR*. This last signal is internal and can not be seen. However, it controls the address multiplexor output (*F* signal). Since the value of the *F* signal is 000001H (upper bits not displayed) and is the same as the intended address, one can infer two things. First, the *AMWR* signal was correctly asserted. Second, the multiplexor module works correctly. To finish the write cycle, the input of the *MAC* signal forces a change from state 1 to state 2. The chip correctly asserts *DV*. Then, a *NAV* signal resets the machine to state 0.

The trailing read access works in a similar fashion. The machine enters state 3 upon detection of the read access. However, this time it does not enable the *EWR* or the *AMWR* signal. Instead, it asserts the *AMRD* and *CAR* signals (besides *EMA*). *AMRD* also controls the address multiplexor output and can be verified by observing the value of *F* (remember that the read address is also 000001H). On the other hand, *CAR* can neither be seen nor deduced at this point. But, a correct assertion of *CAR* would have stored the address 000001H in the current address register (to be kept in mind for later on). To continue with

the read process, a *MAC* is issued changing the machine from state 3 to state 4. Here, *DV* and *OMRD* are asserted. Again, *DV* is external and can be seen, *OMRD* can not. Nevertheless, *OMRD* controls the data multiplexor output and can be verified with the value of the *K* signal. The read value is 001H, which is exactly what the *K* value is. Finally, *NAV* is issued and the states changes from 4 to 5. Notice that state 5 is similar to the initial state 0. This completes this stage analysis. Figure 14 and 15 summarizes the areas tested so far.

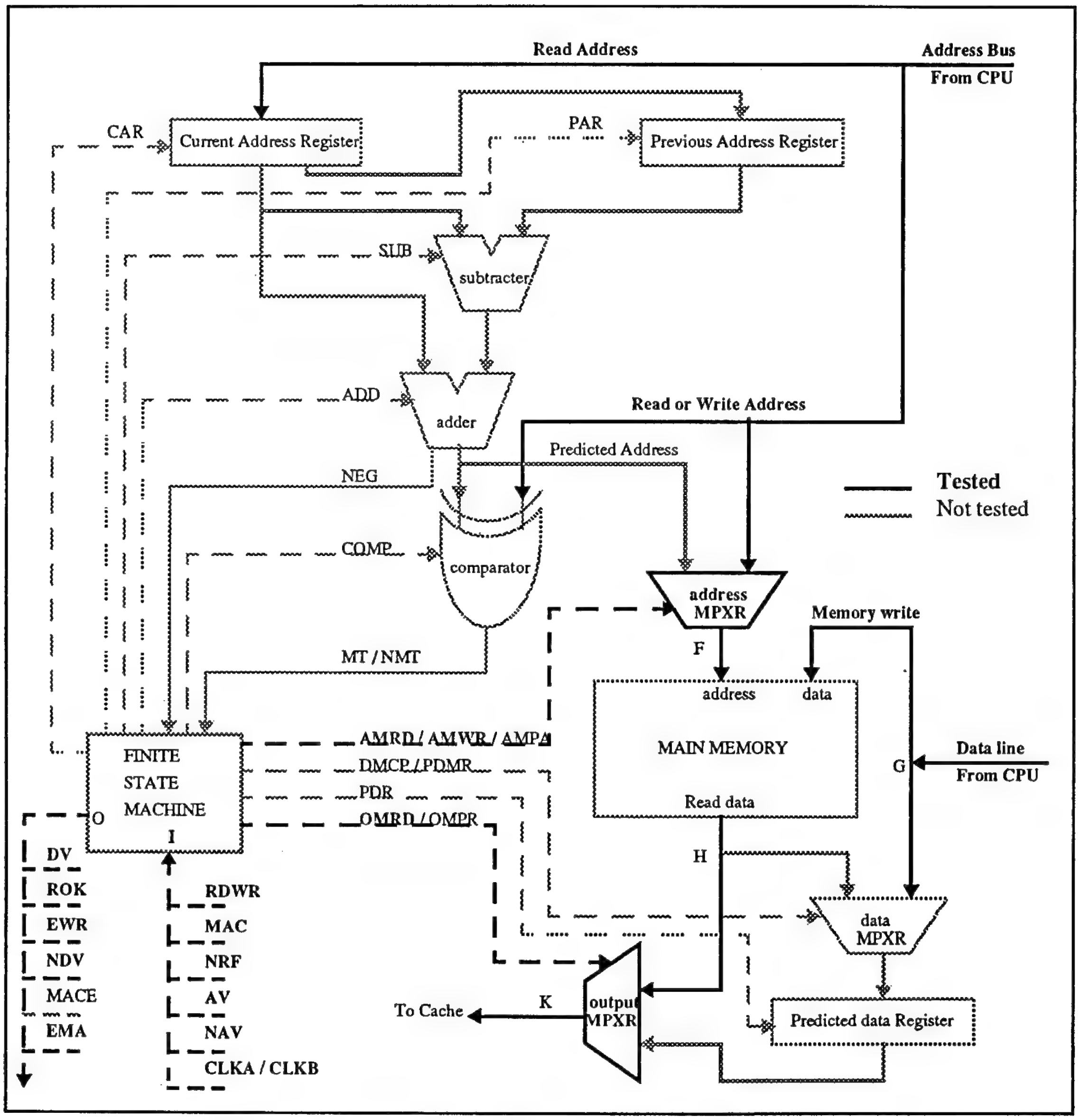



Figure 14: Section 1, Tested Areas of Data Path

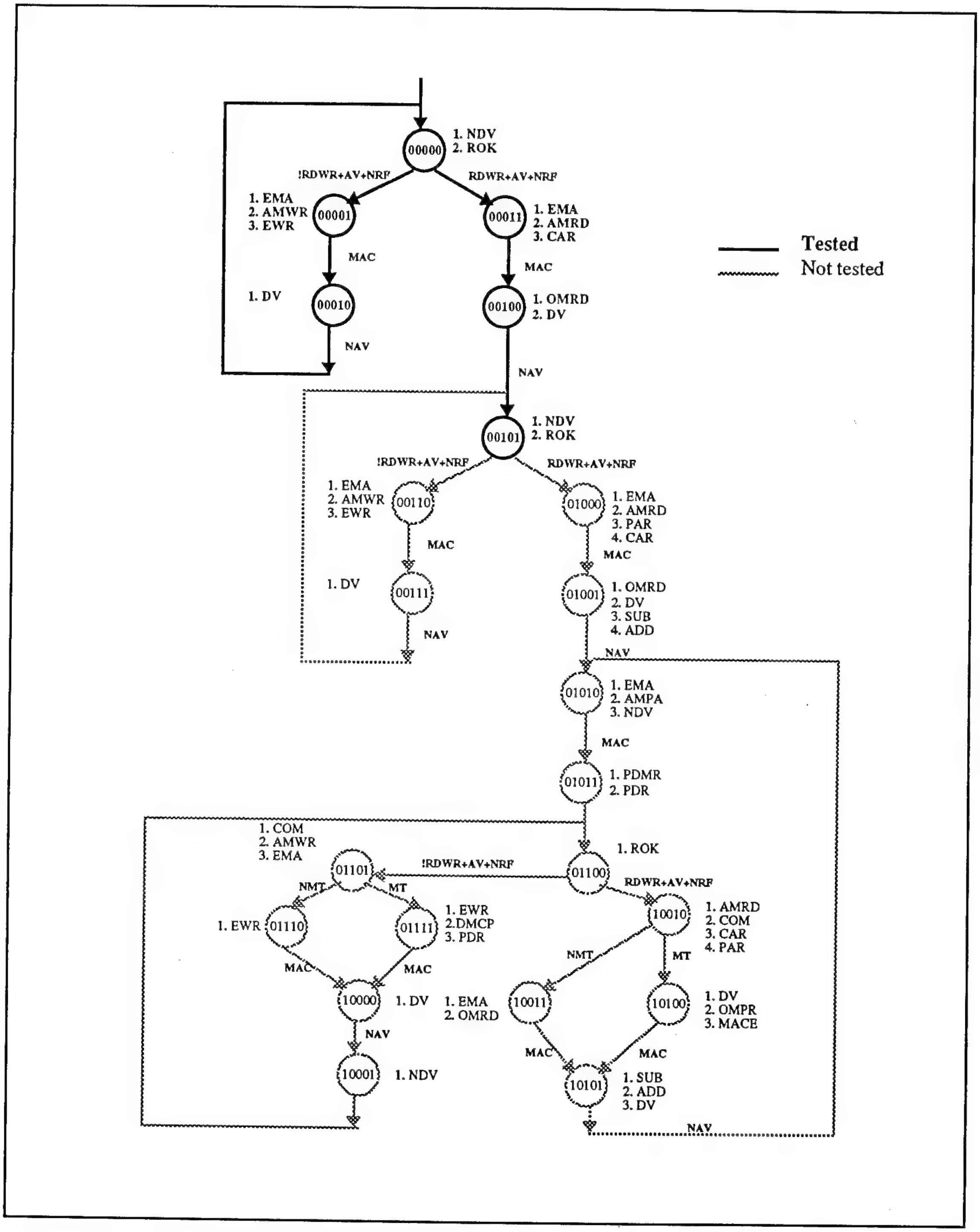



**Figure 15: Section 1, Tested Areas of The FSM**

### c. *Section 2*

The portion of the program corresponding to this section is given by lines 14 through 28. Again, a write followed by a read access is simulated. This time, the value to be written is 006H at address location 000006H. The read access requests data from location 000002H. The value assigned to that memory address is 002H. Both address and value need not be the same. In fact, they are rarely the same. Without loss of generality, this keeps things simple. The captured waveform for this section is presented in Figure 16. As in state 0, state 5 asserts *NDV* and *ROK* to signal that it is ready for input. The write cycle is

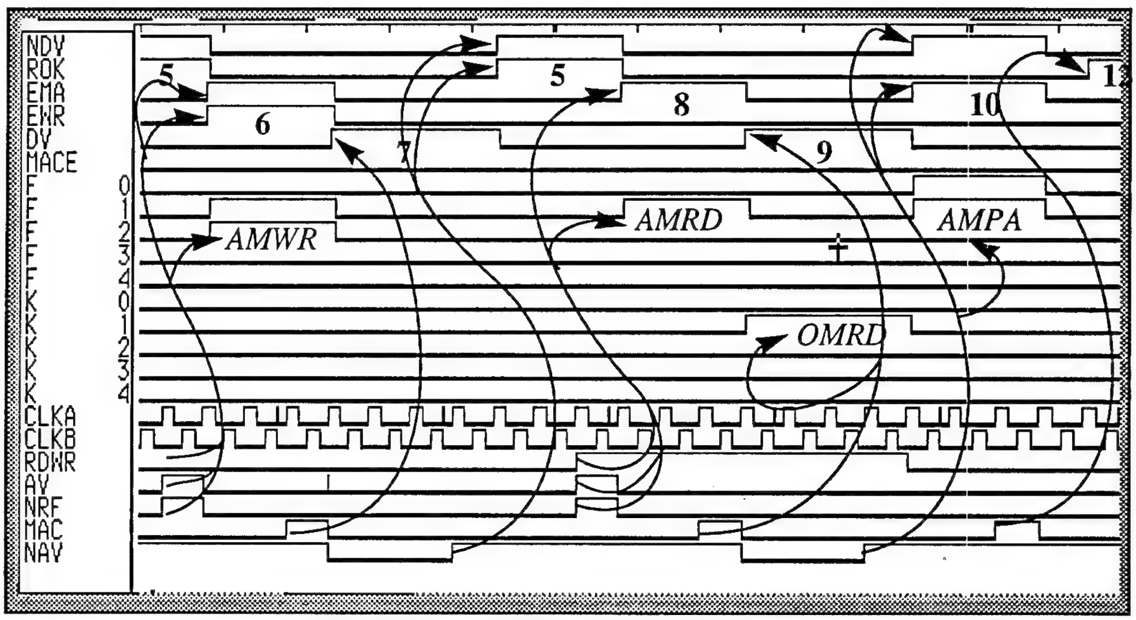



**Figure 16: Section 2, Captured Waveform**

performed exactly the way it was explained in the previous section. The only difference is that state 6 and 7 replaces states 1 and 2. The value of *F* is proof of the correctness of the signals involved. The read cycle is executed a little differently and is explained in the next.

When the second read access occurs, the state changes from 5 to 8. In this state, *EMA, AMRD, CAR* and *PAR* are asserted. The first two have already been checked and

can be verified in the same way. *CAR* and *PAR* are internal and can not be checked in this state. Remember, in the previous section it was assumed that *CAR* was correctly asserted. The same assumption will be made here. Then, the previous address register would hold the first read address and the current register the second one. With this, the chip has enough information stored to make its first prediction. A *MAC* signal comes along and the chip enters state 9. In this state, *DV* and *OMRD* are asserted, as well as *SUB* and *ADD*. *OMRD* is again verified with the *K* value. This time, its value is 002H, which is also correct. The other two internal signals, *SUB* and *ADD*, are the commands for calculating a predicted value. At this point, if everything has gone well, a predicted value of 000003H should exist and a read access to that location should be generated by the chip. When the state changes to state 10 (by the *NAV* signal), the read access is initiated by asserting *EMA* and *NDV*. Also, a new unchecked internal signal is asserted, *AMPA*. This signal selects the other input of the address multiplexor (predicted value). The *F* signal now reflects that value, 000003H. The correctness of this value implies the correct functioning of the Current and Previous register, the adder and subtracter, and all the signals that have been assumed so far. When data at the requested location is ready, the *MAC* signal is asserted, forcing a change to state 11. *PDMR* and *PDR* are asserted to latch the read value into the Predicted data register. The read value is 003H, but it will change when a write through operation is executed. For now, these two signals can not be checked. Without the need of any input, the state changes to state 12, which is like states 0 and 5. Notice that *ROK* is the only signal asserted since *NDV* was asserted in state 10 and negated in state11. This is a clear landmark that state 12 has been reached. This completes the analysis for this section, as shown by Figure 17 and Figure 18.

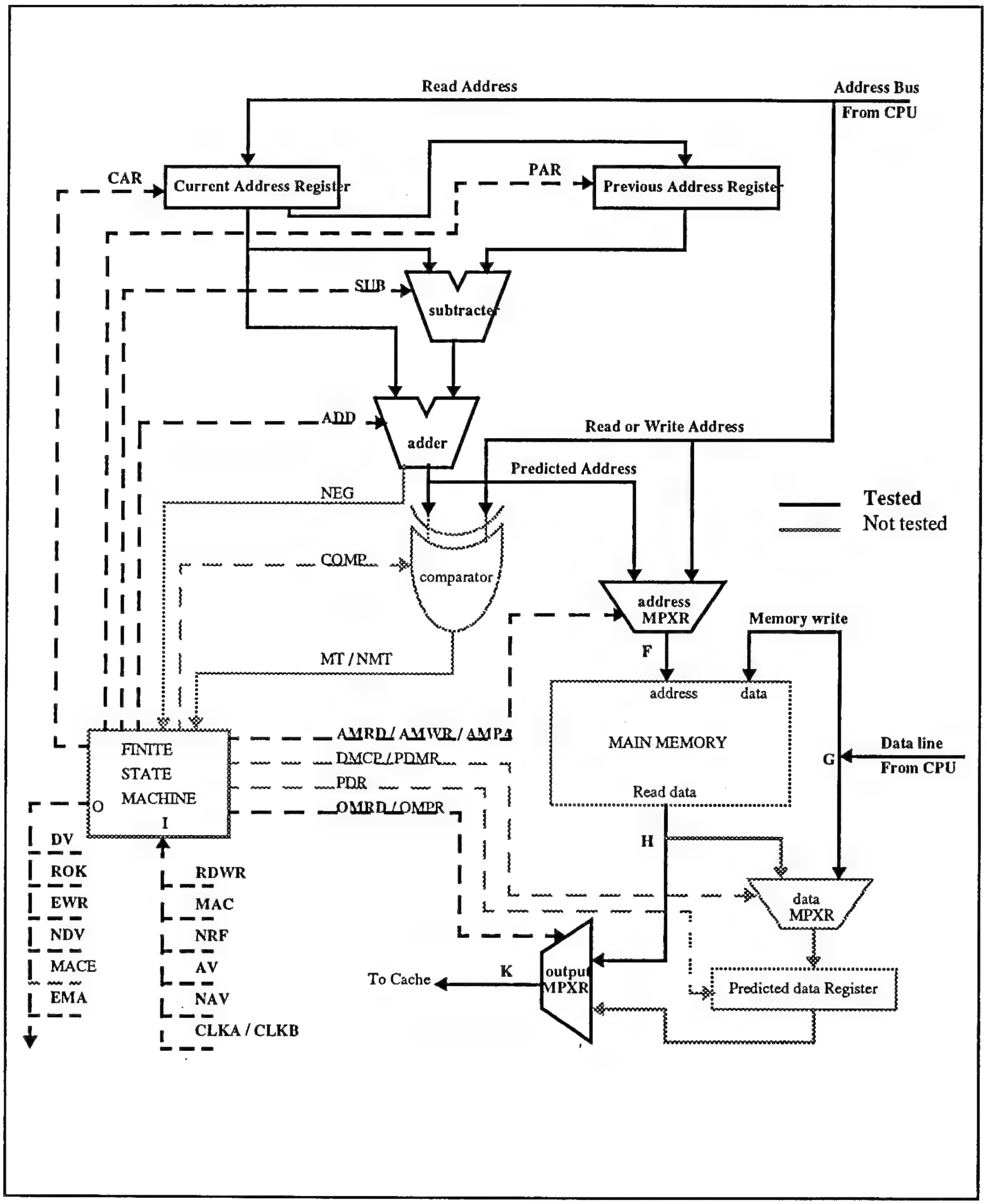



Figure 17: Section 2, Tested Areas of Data Path

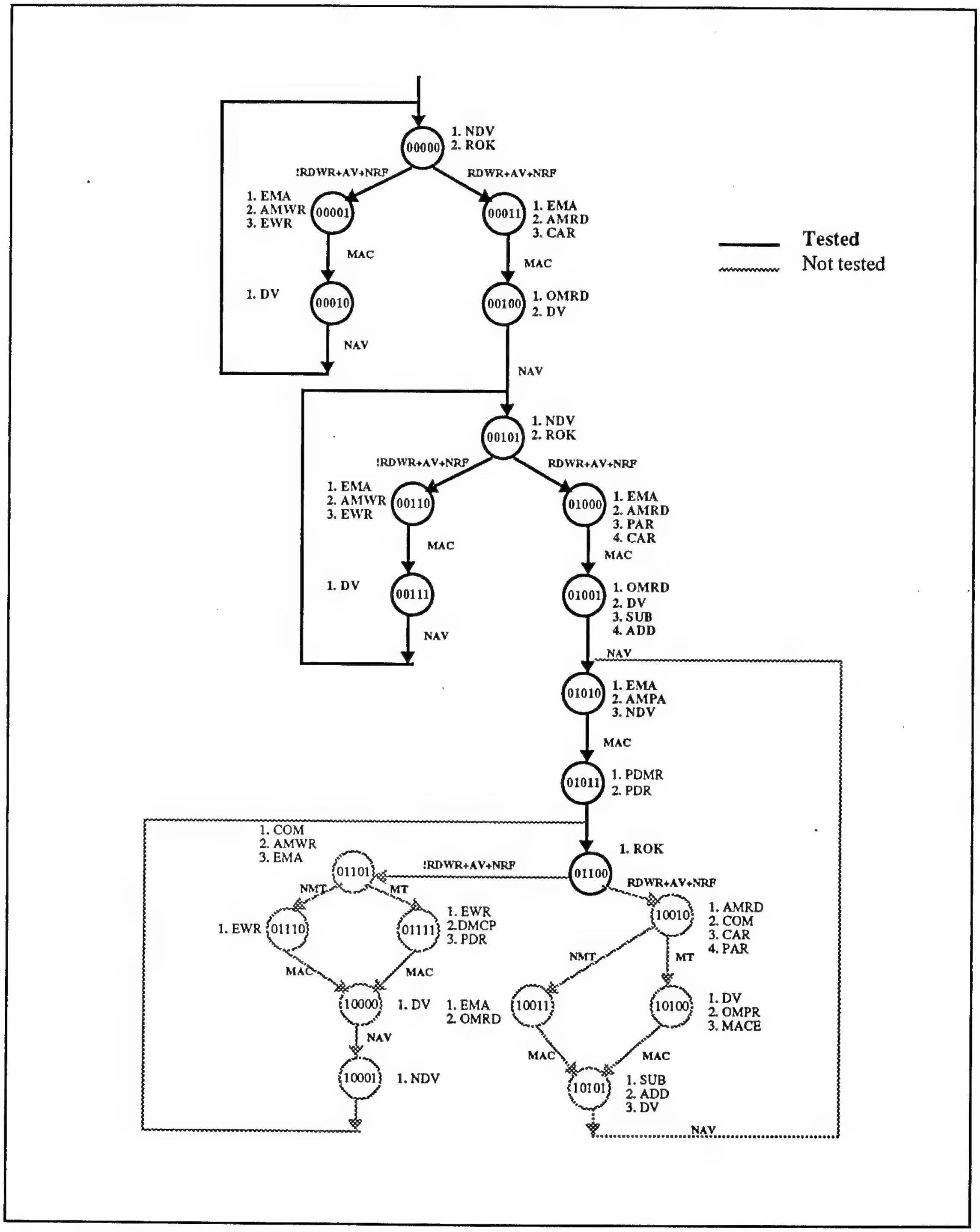



**Figure 18: Section 2, Tested Areas of The FSM**

### *d. Section 3*

At the program level, lines 29 through 44 simulate another write followed by a read. Address 000003H is accessed and written, the value is 00FH. Later, the read access requests data at the same location. Remember from the previous section that the predicted address 000003H exists and that data at that memory location has (hopefully) been latched in the predicted data register. Figure 19 presents the corresponding waveform diagram. When the write access occurs, the state changes from 12 to 13. There, a memory

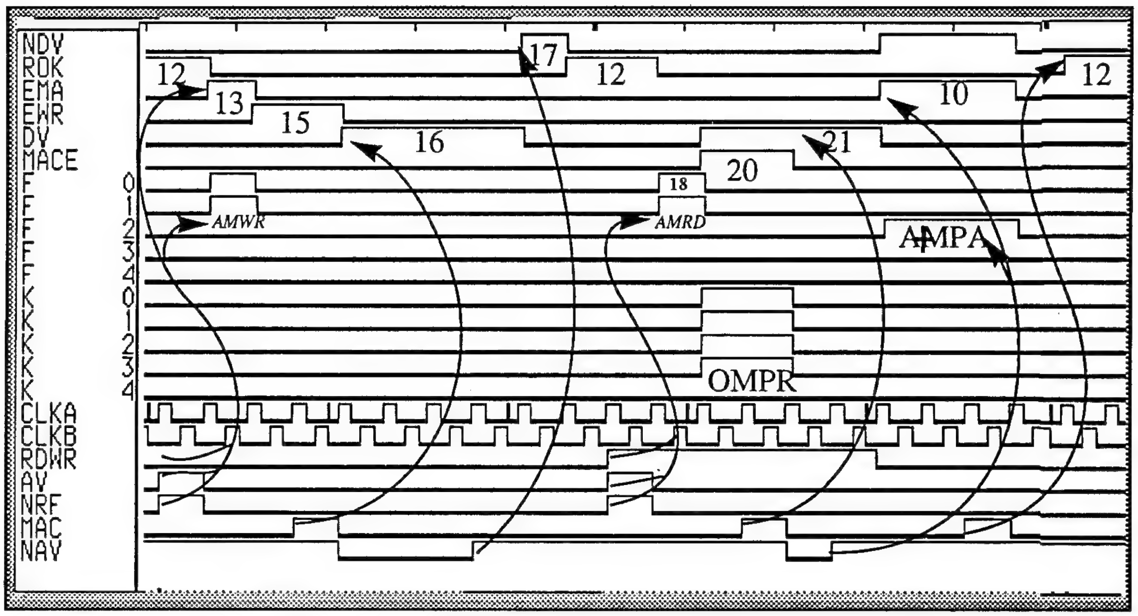



**Figure 19: Section 3, Captured Waveform**

access is enabled with the assertion of *EMA* and the address is selected with the *AMWR* signal. This time, a compare command is given (*COM*) because the buffer needs to maintain coherency in its data. In fact, the requested address matches the predicted one. At this point, either state 14 or 15 could be entered, depending on the output of the comparator. These two states happened to have the same external signals but different internal ones. State 15 has *DMCP* and *PDR*, used to replace the stored value. For this particular case, state 15 should have been entered and the stored value replaced with 00FH. The only way to determine

correct operation is to let the write cycle finish and make a read access to address 000003H. This written value should be provided by the chip. This is exactly what follows. The write access finishes in the same manner as was explained, cycling through states 16,17, and back to state 12. When the read access arrives, the state moves to 18. *AMRD* is asserted (the read address is reflected in the value of the *F* signal) and a compare command is given. The transfer among registers is also performed. The assertion of *DV* and the value of *K* indicates that the comparator successfully found a match and state 20 was entered. Moreover, the write through operation also worked (the *K* value is 00FH vice 003H). Here, the *MACE* signal is asserted for the first time. Notice that this is the only state in which that signal is asserted (another clear landmark). The read cycle proceeds in the same way explained in section 2; state 21 calculates the new predicted value (000004H) and then states 10,11, and 12 initiates and completes a read access for that address (see the value of *F* in state 10). Figure 20 and 21 shows the sections tested in this analysis. With the exception of the *NEG* signal, the data path has been completely checked and found to be fully working. These findings are reinforced in section 4 and section 5.

### *e. Section 4*

This section was programmed to test the two missing states, 14 and 19. The program lines 45 through 60 simulate a read followed by a write. The read access is to address 000005H which holds the value 005H. The write is done to address 000006H. The current predicted value is 000004H, so when the read access arrives a no match should occur and state 19 should be entered. Figure 22 presents the corresponding waveform diagram. Notice that *EMA* is asserted and not *DV* or *MACE*. This is a clear indication that state 19 was entered and that the comparator correctly performed its function. The rest of the cycle continues the same way it was explained in the previous section. This time though, the predicted value should be 000007H. The reader is reminded that the last two consecutive read accesses have been 000003H and 000005H, which have a displacement of two. Adding this to the most recent address gives 000007H. This result shows up when

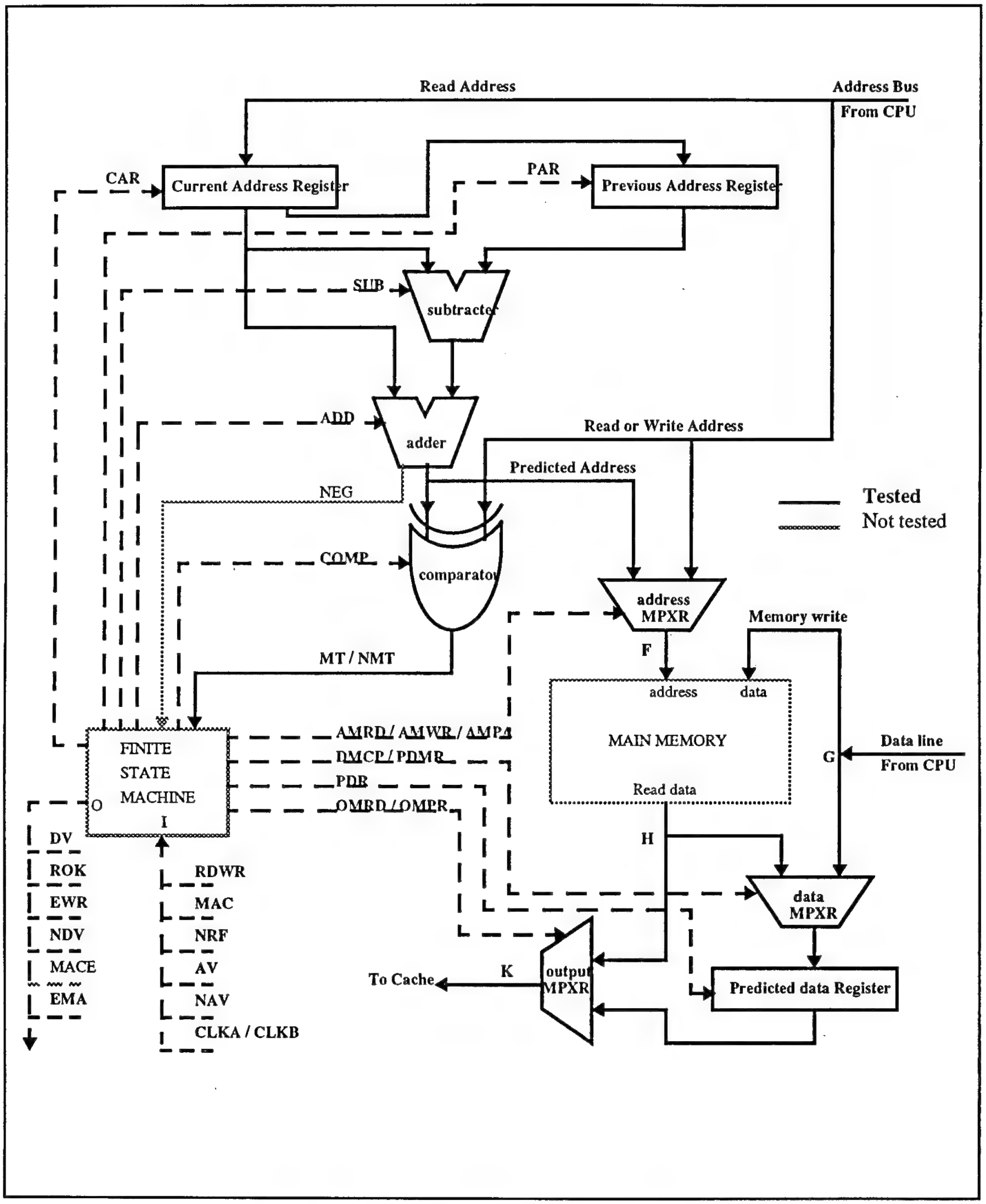



Figure 20: Section 3, Tested Areas of Data Path

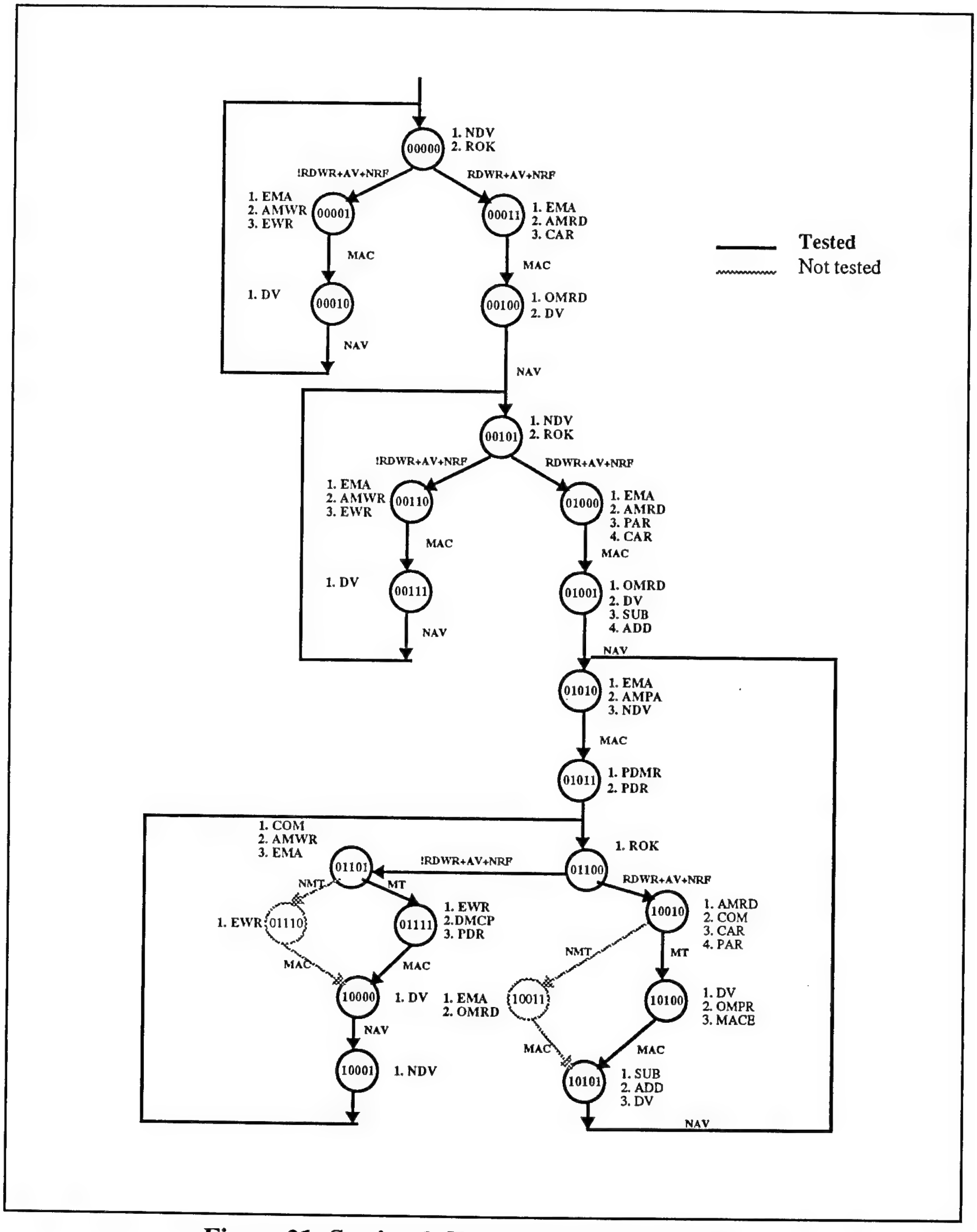



**Figure 21: Section 3, Tested Areas of The FSM**

the chip initiates a read cycle to that address (AMPA). State 14 is entered when the incoming write accesses an address other that the predicted one. The visible output of state 14 is the same as that of state 15. Here, the testing relies on the correct functioning of the comparator.

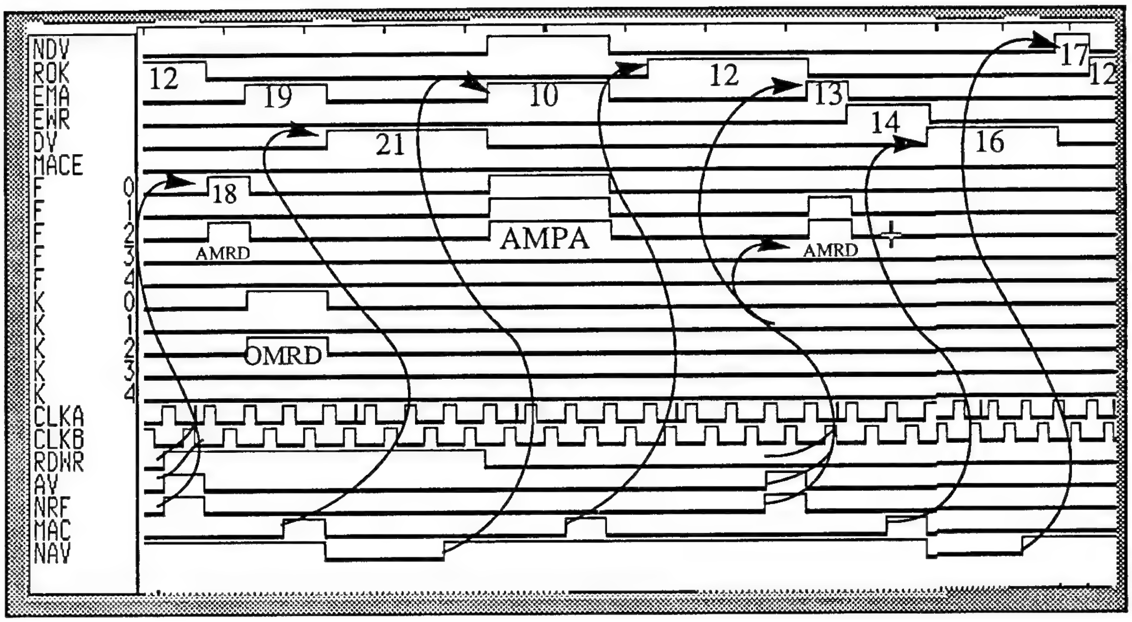



Figure 22: Section 4, Captured Waveform

### *f. Section 5*

This section is basically a repetition of section 3 to confirm correct operation of the adder, subtracter and comparator. The last lines of the program simulate, once more, a write followed by a read (both to address 000007H). The captured waveform is shown in Figure 23. The write access changes the value stored in the predicted data register. The value written is 007H and is shown as the *K* value when the read occurs. Again, notice the landmarks, *MACE* is asserted in state 20 and *DV* is very wide since it asserted in both state 20 and 21. Furthermore, the predicted value is now 000009H since the displacement is still two. A last thing worth mentioning is that the end of the waveform does not terminate in state 12. As a matter of fact, states 11 and 12 were not executed this

time because the program failed to provide the last *MAC* signal. This completes the functional testing of the chip. Other tests of interest were also performed and are explained next.

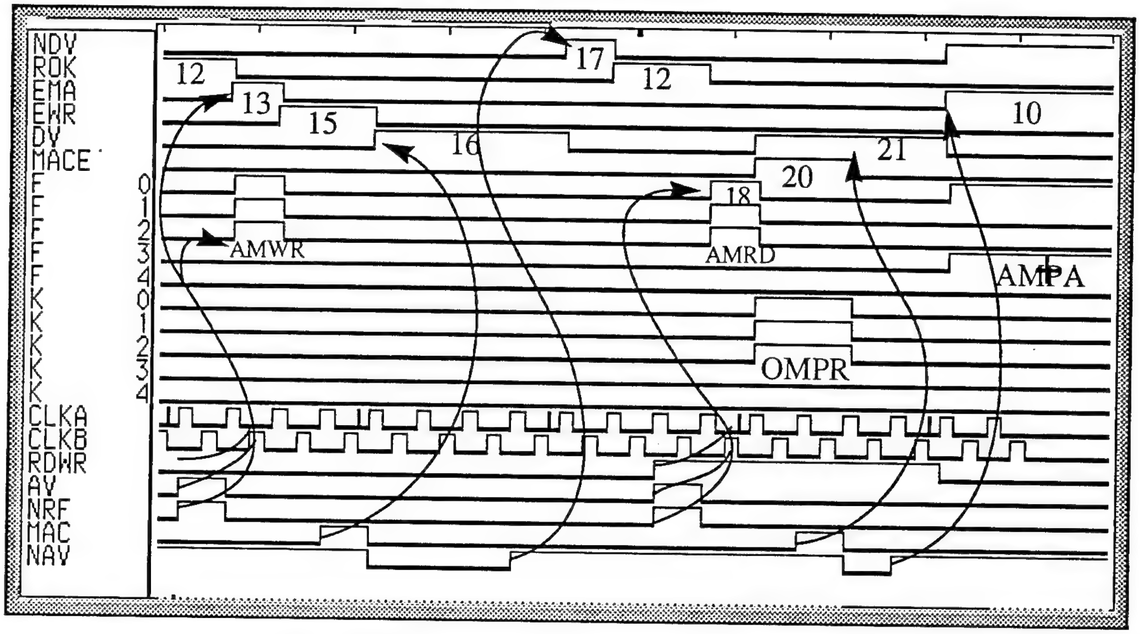



**Figure 23: Section 5 Captured Waveform**

## 4. Other Tests

Other tests were performed in order to find out different parameters of interest. The following sections list these parameters and presents obtained results.

### *a. Failing Rate*

The clock signal for the finite state machine is provided by signals *CLKA* and *CLKB*. Both signals are a vector of inputs in which a logic "1" is maintained for two clock periods and logic "0" is maintained for four clock periods. *CLKB* lags *CLKA* by three clock pulses. Appearing as this:

*CLKA* 110000

*CLKB* 000110

The machine was tested at various clock rates. It was found that the chip best worked at a clock rate of 2 MHz (a period of 500 ns) with a power supply of 5.0 volts. The maximum failing rate detected was at 5 MHz with 3.0 volts input and at 10 MHz with 5.0 volts.

### *b. Latchup*

Before the chip was functionally tested, this test was performed. The aim was to verify that the chip did not suffered from the parasitic effect of latchup. The result was, in fact, negative. The procedure used was simple, an ammeter was connected in series with the voltage supply. The voltage was then raised in increments of 0.5 volts and the drawn current was recorded. Table 1 presents this results. The obtained values are clear enough and need not be plotted. If there had been a latch up problem, the current would have increased considerably around 4.5 volts.

**Table 1: Latchup Test Results**

| Voltage | µAmp |
|---|---|
| 0.5 | 0.00 |
| 1.0 | 0.00 |
| 1.5 | 0.01 |
| 2.0 | 0.02 |
| 2.5 | 0.03 |
| 3.0 | 0.04 |
| 3.5 | 0.06 |
| 4.0 | 0.09 |
| 4.5 | 0.12 |
| 5.0 | 0.15 |

### c. *Noise Margins*

The HP10348A 8-Channel CMOS Tri-State Buffer Pod was used to buffer the TTL outputs of the HP16520A Pattern Generator, providing CMOS level input to the control pins of the RPB chip. The noise margins were found experimentally by measuring with an oscilloscope, the voltage threshold of the outputs of the RPB. The thresholds for the Buffer Pod were obtained from its reference manual. Figure 24 depicts graphically the results obtained. With these values, the calculated margins were; NML= 0.3v, NMH=1.8v.

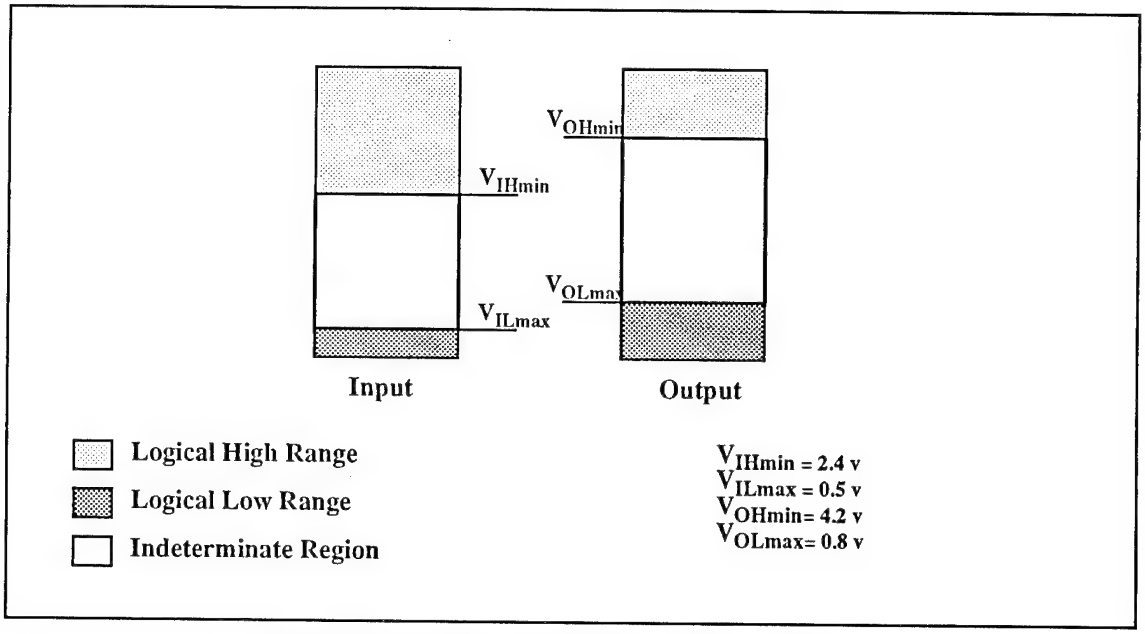



**Figure 24: Measured Voltage Thresholds**

### d. *Power Dissipation*

The Chip's power consumption was calculated to be 45 mW at normal operation of 5 v and 2 Mhz. The drawn current measured was 9 mA. However, this measurement does not reflect the worst case dissipation which occurs whenever three "ones" are being added in one cell.

# III. FUNDAMENTAL BLOCK DESIGNS OF THE PRC

## A. SYSTEM OVERVIEW

It was mentioned in Chapter I that the sole goal of the redesign process was to grant snooping capability and to give the ability to track multiple address traces to the Prediction Buffer IC. Figure 25 shows this idea in a simplified form. From the global perspective, each line of the PRC operates similarly to the RPB. A line is selected to generate a predicted address for a particular displacement. When a read miss address is received, the address is compared against all the predicted addresses. The comparison is performed in parallel by all lines. The line with a "match" becomes active and passes its stored data to the interface unit which is in charge of performing address snooping and data transfer. A "no match" is an indication of a displacement change and a new line is selected to be the active one. Associated with line selection is line replacement. When all lines are full (contain predicted addresses) and a new predicted address is generated, one of the lines needs to be replaced. Most of the suitable known replacing algorithms are expensive and complex to implement in hardware. Because of that, The PRC implements a somewhat modified version of the Second Chance algorithm which is reasonably effective and easy to implement [Ref. 11]. The algorithm is basically a FIFO List in which each member of the list has an associated flag. When a new member arrives, it is added to the tail of the list and the member at the head is removed. If the flag of the removed member is set, the member is again added to the tail and the new head member is removed. The process is repeated until a head with a clear flag is found. The flag is always cleared when the member is added to the tail and it is dynamically set when the member has recently been used.

As in the RPB IC, the PRC needs to maintain data coherency at all times and at all lines. The chosen method disposes of the stored data if a match is found during a write access. The reason behind this is that the interface unit was designed to interface with the PowerPC-603 CPU which performs single beat or burst read/write data transfers. A single

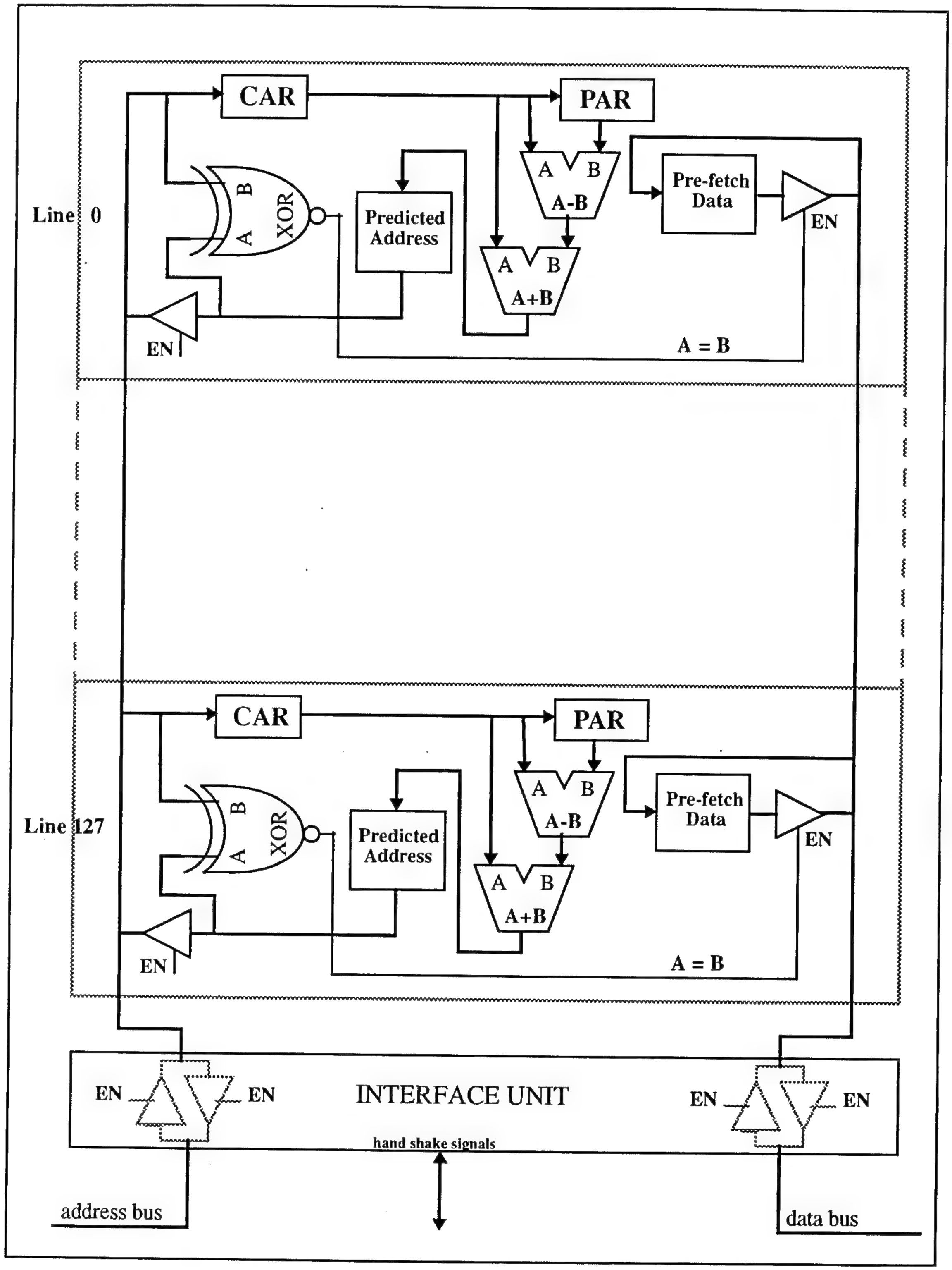



**Figure 25: PRC Functional Block Diagram. After Ref. [2]**

beat is a noncacheable operation in which 1 through 8 bytes are transferred during an access. A burst operation transfers 32-bytes and is cacheable (it actually fills a line of 32-bytes in the on-chip data cache)[1]. The PRC is only concerned with burst operations, therefore it stores 32-bytes of data for a predicted address aligned to a double-word boundary (four bytes per word). A single beat read access can be ignore with no harm. Ignoring a single beat write operation could cause data corruption. A process of updating 32-bytes at a time (a burst write) can easily be implemented. However, updating 1 through 8 bytes can be complicated and expensive since their addresses could be misaligned to the double word boundary. The safest policy is then to flush the data if a hit occurs during a single or burst read access.

Although the presented block diagram conveys the idea well, it is far from representative of the actual architecture of the PRC. The diagram implies that many modules (lines) perform the same set of functions. That is not optimal.To reduce cost and complexity and increase efficiency, the architecture used implements functions for all modules. An analysis of the required functionality dictates that the chip must be capable of performing the following functions:

- Snooping,
- Predicting,
- Line Replacing,
- Storing and Detecting,
- Data Updating or Flushing
- Data Transfer and Flow Control

The structure of the architecture used is composed of six modules, each implementing one of the listed functions. Four of these modules have been designed and implemented. The following sections explain, at the block and functional level, these four modules. They are first explained in isolation and then it is shown how they interact and communicate with each other. Their implementation in hardware is given in the following chapter.

1. The transfer is performed in four cycles (beats) of 8 bytes (64 bits) each.

## B. THE SNOOPING MODULE

This module was designed to snoop read/write operations between the IBM-Motorola PowerPC-603 Central Processing Unit (CPU) and the Main Memory system. The module main function is to identify a valid read or write access and provide appropriate acknowledgment to the cpu if a read hit occurs. In addition, it alerts the other modules when a valid access is detected. Figure 26 presents this unit. It is composed of an address parity checker, a D-flip flop register with clear, and a finite state machine. The parity checker checks the incoming address with the parity bits provided by the cpu. A parity error signal is sent to the finite state machine which inspects this signal only if a potential access is detected. A potential access happens when the Transfer Start signal (*TS*) is asserted by the cpu. The FSM must qualify this assertion before granting valid status. A valid status is

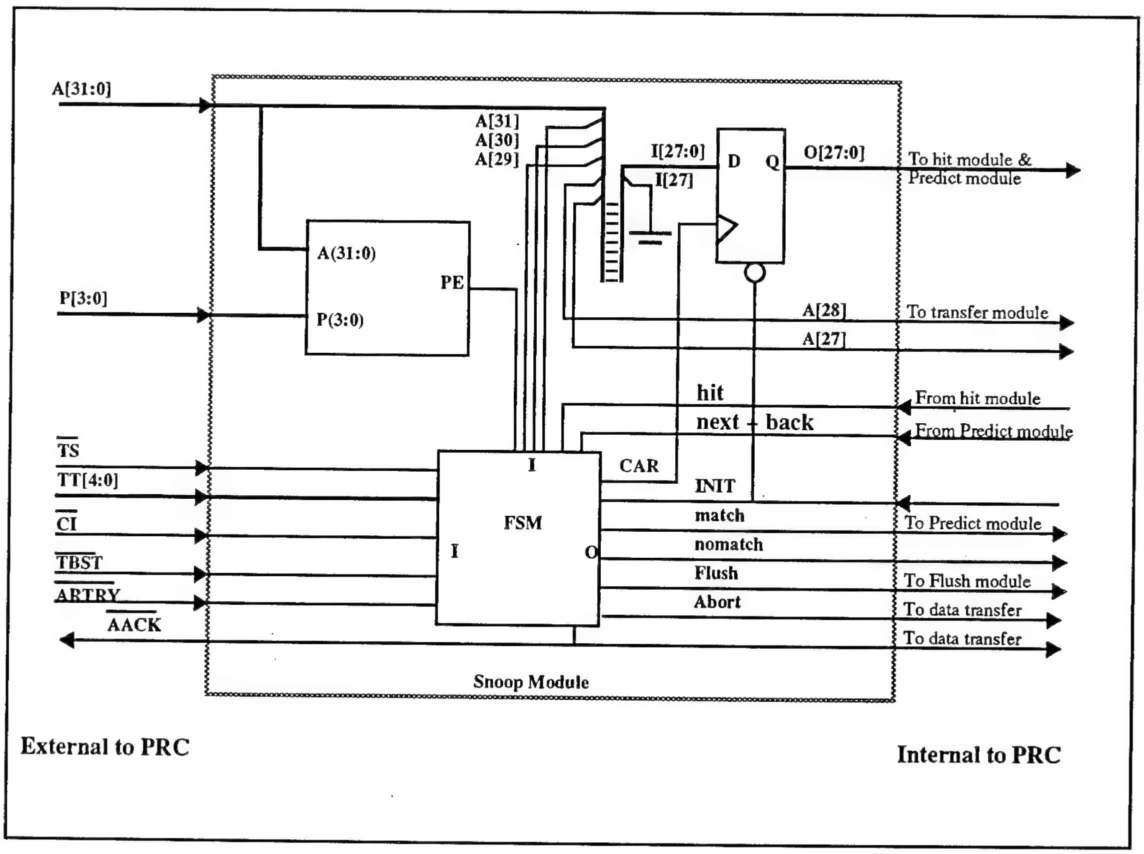



**Figure 26: Snoop Module Functional Block Diagram**

obtained when no parity error exist and either a burst read access occurs or a write access occurs (single beat or burst). If a parity error exists, the module aborts the operation and snoops for the next potential access (the unit was not provided with a retry signal to avoid all possible interference). Otherwise, the FSM latches the requested address into the D-register (Current Address Register) which provides input to the hit detection module and the predict module. Notice that only the first 27 bits, A(26:0), of the physical address are latched. Bits A[29]-A[31] are fed to the FSM and bits A[27], A[28] provide additional information meaningful only to the data transfer module. The added bit, I[27], acts as a flag which differentiates between an existing and non-existing address (0 for existing, 1 for nonexisting).The FSM determines whether the access is a read or write by decoding the five bits of the Transfer Type signal (*TT*) (The different types of operations are listed in table 9-1 on pp. 9-11 of [Ref. 10]). In addition to this signal, for a burst read access, the three most significant bits of the address need to be a logical zero, the Cache Inhibit signal ($\overline{CI}$) must be negated, and the Transfer Burst signal ($\overline{TBST}$) must be asserted. If all these conditions are met, the module inspects the *hit* signal and alerts the predict module with a *match* or *nomatch* signal. If a hit is found, the module also sends an acknowledgment signal (*AACK*) to the CPU indicating the termination of the address transfer and to the memory system to stop the current read access. The same signal alerts the transfer module for the initiation of the data transfer. The module then waits for a *next* or *back* signal to reset to the snoop position. While waiting, if the Address Retry signal (*ARTRY*) is asserted by the memory system, the module raises an abort signal. This is done because the Retry Signal will also be received by the cpu which will abort the operation upon assertion. Similarly, for write access, the module inspects the *hit* signal and asserts the *flush* signal if a hit is found. The FSM unconditionally resets to the snoop position. This completes the operation of the snoop module. For detailed information concerning the 603 signal description or address bus operattion, the reader is referred to chapters 9 and 10 of [Ref.10].

## C. THE HIT DETECTION MODULE

This module was designed to hold all the predicted addresses and to simultaneously compare all those addresses against the incoming requested addresses. In addition, the module provides the line number at which a match is found. Figure 27 presents the block diagram for this module. The unit utilizes a D-flip flop register and a comparator per line. In each line, the register holds a predicted value for a particular displacement and the comparator compares that value against the incoming request address. The requested

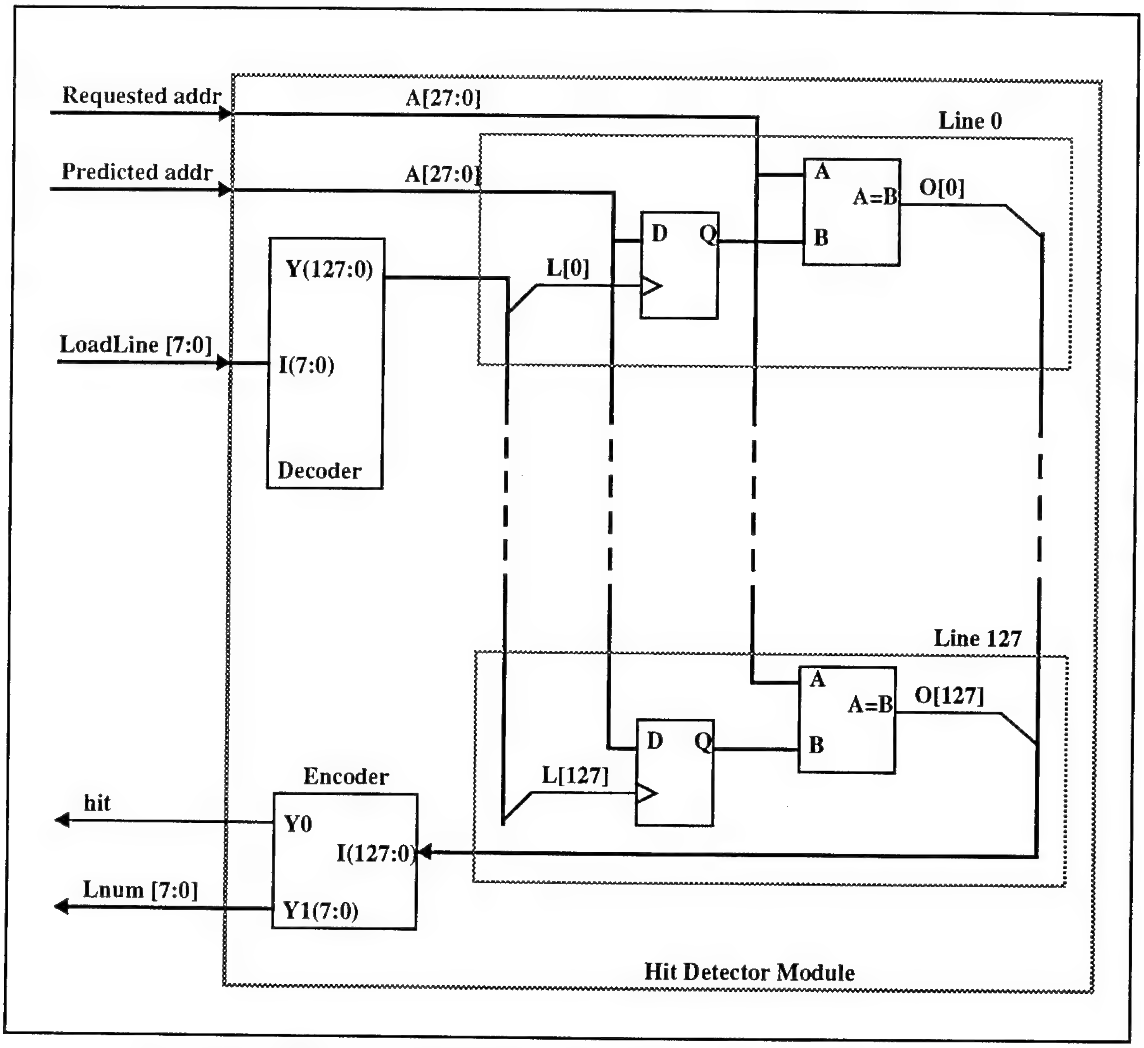



Figure 27: Hit Detection Module Functional Block Diagram

address changes every time the Current Address Register is loaded by the snoop module. If both values are the same, the comparator asserts its single bit output (1 for equal, 0 for not equal). The comparator recognizes an empty register or a existing non address using the setting of the most significant bit (A[27]). A requested and a predicted address will always have that bit set to zero. Therefore, if a register holds a non predicted address, the MSB will be set to 1 and the comparator output will be zero for that line. A hit is detected by the assertion of any of the comparator outputs. The line number at which a match is found is computed from the position of the asserted line on the Out bus (O). A priority encoder is used for this. A standard encoder will not be able to handle the special situation in which two or more lines generate the same predicted address for two (or more) different displacements[2]. The priority encoder takes care of this case by selecting the line with the highest priority and disregarding any other lines (lines with higher numbers have priority over lower ones). As an alternative, the chip could have been designed to prevent such a situation. However, the gains of doing so do not justify the increase in hardware cost. Finally, a register is selected and loaded by specifying its line number with the *Loadline* signal. A decoder component is used to decode the line number and assert the appropriate register clock line. The *Loadline* signal is 8 bits wide, seven of which are use to decode 128 positions. The most significant bit (*Loadline*[7]) is used as an enable bit. When set to 0, the output of the decoder is zero in all 128 lines. When set to 1, the decoder selects and asserts one of its 128 outputs depending of the binary value of its lower 7 input bits. This concludes the functioning of this section.

2. Consider the following example: Two consecutive read access arrive and are stored at line 1. Suppose the first read is to address 001 and the second one is to address 004. This generates a predicted address of 007 (displacement is 003). Now a third read access occurs to address 005. Since it is a no match, it is stored in a different line, say line number 2. When the fourth read access occurs at address 006, a predicted address of 007 is also generated for line 2 (displacement is 001). If the fifth read access is to 007, both lines 1 and 2 will find a match and both will assert its comparator output. A normal encoder will fail under this situation.

## D. THE PREDICT MODULE

The main module function is to generate a predicted address and to specify the time at which a line is replaced. Figure 28 presents the functional block diagram for this unit. It uses a register file to store the set of Previous Addresses, an adder to calculate a predicted address, and a finite state machine to control the sequence of operations. The input to the register file comes from the Current Address register. Notice that the flag bit A[27] has been removed and is not passed into this module. The adder implements equation 3.1 (which is equation 4 of figure 2).

$$\phi = 2*\beta + (-\alpha) \qquad \text{(Eq 3.1)}$$

The multiplication term is obtained by adding a zero to the LSB of the B input. The second term is obtained by providing the A input (register output) in 2's complement form (That is the reason for inverting the input of the register file and providing a carry in bit). The 2's complement is formed in the adder when it adds the inverted input with the carry

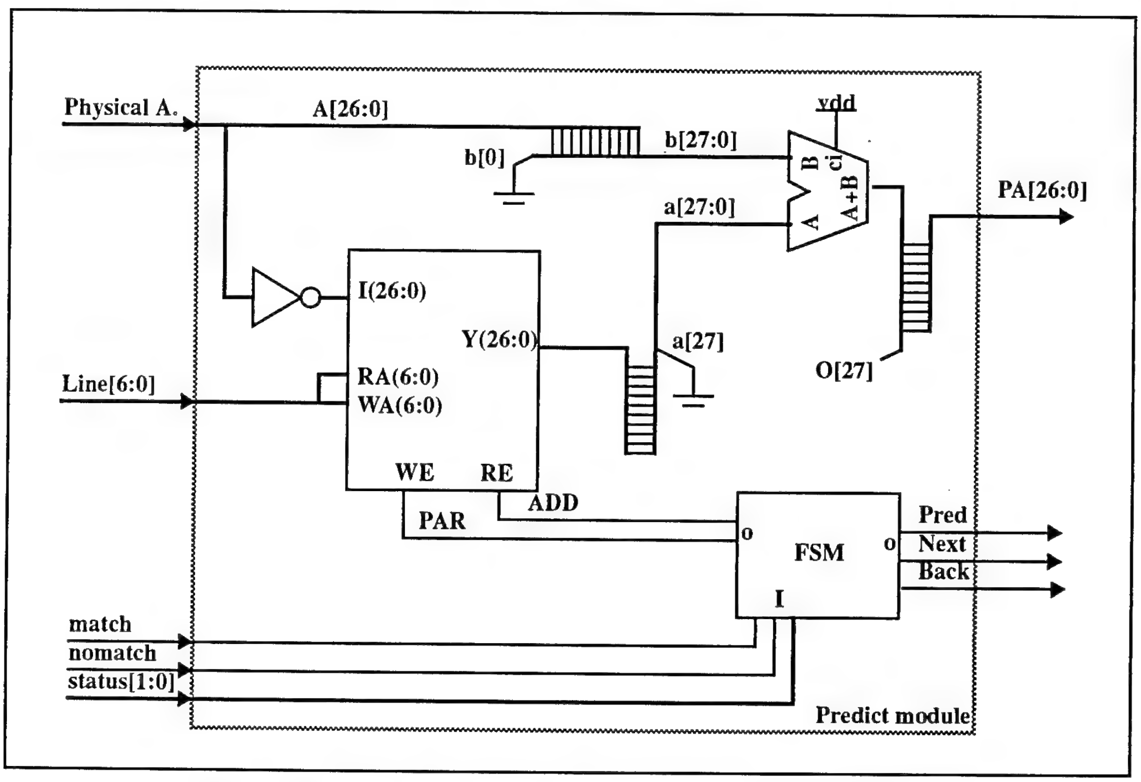



**Figure 28: Predict Module Functional Block Diagram**

in bit. Notice that the MSB of the output of the adder has been stripped off, the predicted address is given by bits (26:0). This configuration works nicely because no overflow or negative number is generated (the address wraps around the ends). The finite state machine has to decide when to store an incoming address into the register file, when to replace the line number, and when to predict an address. The FSM uses the *nomatch* signal to determine whether a displacement change has occurred. It keeps track of two consecutive *nomatch* signals. When the first *nomatch* signal is received, the FSM inspects the *status* signal to verify that the line number presented by the *Lnum* signal is valid. If it is, the address is latched into the register (*PAR*) at the current line number. If is not, the FSM waits until the valid status is obtained, then latches the address. The line number is maintained and will not be changed until this FSM sends the *next* signal to the replace module (this is sent only after the second no match occurs). When a second *nomatch* occurs, the state machine reads the previously stored value (*ADD* signal) and the adder performs the prediction. The FSM waits for the result to come out of the adder, then asserts the *pred* signal to inform the data transfer module that the predicted value is valid. The FSM machine then latches the incoming address into the file register at the same location where it just read. The current address now becomes the previous address for that line (this is why only one current address register is needed). The FSM terminates by sending a *next* signal to the replace module for a new line number. On the other hand, if a *match* signal is received, The FSM checks the *status* signal and waits until the replace module places, into the *Lnum* signal, the line number at which a match was found. The value stored at that line number (previous already exists if a match occurs) is then read and a new prediction address is generated for that line. The FSM sends the *pred* signal to the data transfer module. This time, however, it does not send a *next* signal to the replace module. Instead, it sends a *back* signal to the replace module indicating that the line number displayed prior to the match line number should be restored. Last, if a *match* signal occurs in between two consecutive *nomatch* signals, the FSM performs the match operation as described and returns to the original place where it was interrupted. Although the assertion of a *match* and

*nomatch* signal will not overlap, the FSM will ignore both inputs if it happens. This concludes the functioning of this module.

## E. LINE REPLACEMENT MODULE

This module implements the modified version of the Second Chance algorithm. It's main function is to generate a line number and to switch, when a hit occurs, to the line number provided by the hit detection module. Figure 29 presents the block diagram for this module. The module uses a binary mod 128 counter with a flag associated with each count,

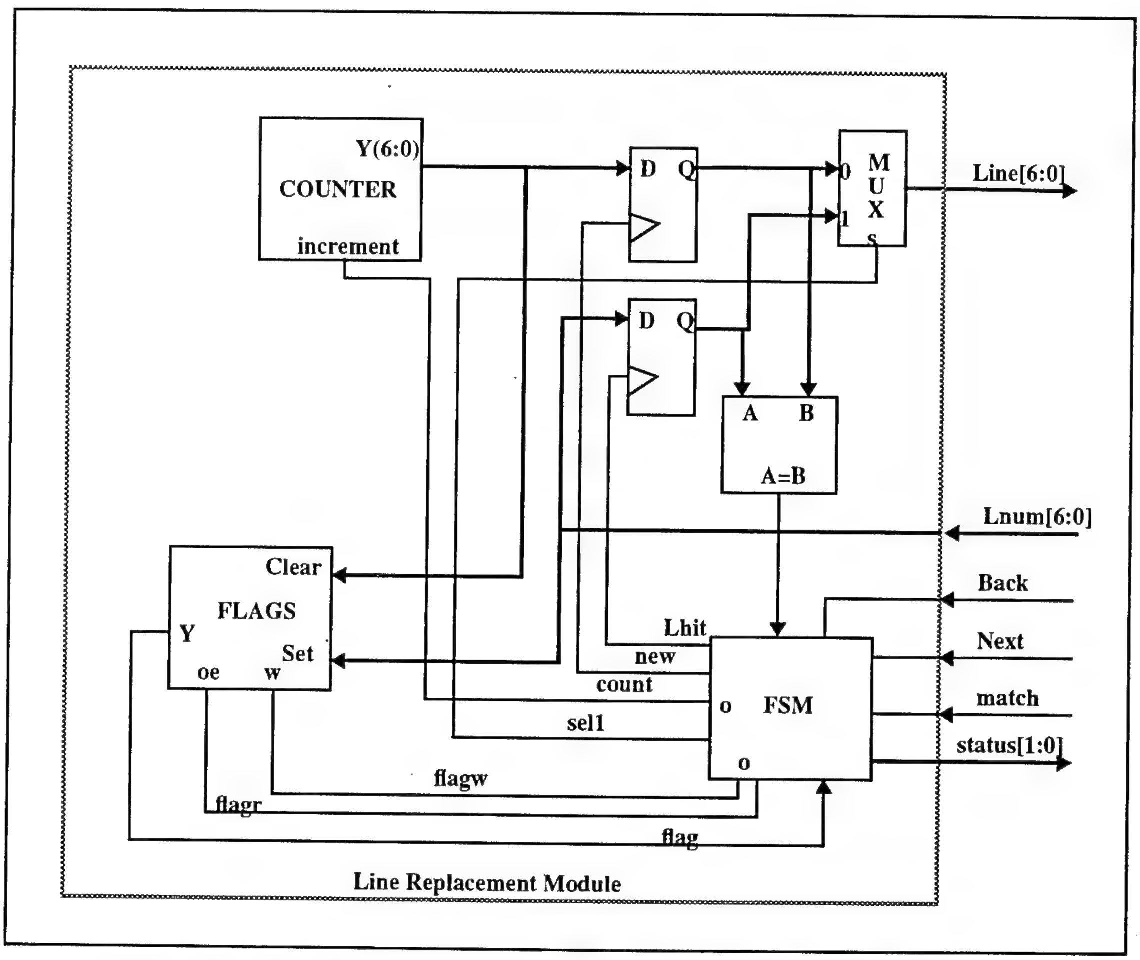



**Figure 29: Line Replacement Module Functional Block Diagram.**

two 8-bit wide registers, a mux, a comparator, and a finite state machine. When the FSM receives a *next* signal from the predict module, it latches the counter's output into one of the registers and the register is selected (mux) displaying the current line number. At this point, a valid status signal is displayed by the FSM. While the current line number is being used by the predict module, the FSM determines what the next line number will be. To do this, the FSM increments the counter and inspects the associated flag. If the flag is set (logical 1), the flag is cleared and the counter is incremented one more step. The processes is repeated until a count with a clear flag is found. At this point, the FSM goes to the idle position where it is ready to receive a *next* command. Notice that the advantage of looking ahead is that when a new line number is needed, the module's response time is ($T_{dff} + T_{mux}$) which is a very small delay. When a hit occurs, the module receives a *match* signal from the snoop module. Upon assertion, the FSM displays a not valid status, waits for the line number be provided by the hit module, then latches the number into the other register and selects that register. A valid signal is then displayed. The flag associated with this number is set. While this hit number is being used, the FSM compares the contents of both registers. If they are the same, the FSM replaces the content of the first register with the next line number (coming out of the counter) and proceeds with the increment-inspect process. If the contents of both registers are not the same, the FSM waits for a *Back* signal from the predict module to restore the line number that was displayed previous to the hit. It is perhaps appropriate to explain why the FSM needs to compare the contents of both registers and change one if they are the same. The reason behind this is that when the hit line number is displayed, the predict module will generate a predicted address for that line and then will send a *Back* signal to restore the interrupted line number. The replace module will switch registers. Since the contents of both registers are the same, the same line number will be restored and the predict module will override the just predicted address when the following read access occurs. Therefore, to avoid this situation, the restored number is changed in the event that both line numbers are the same. To conclude the chapter, Figure 30 presents how each module communicates and interacts with the rest of the system.

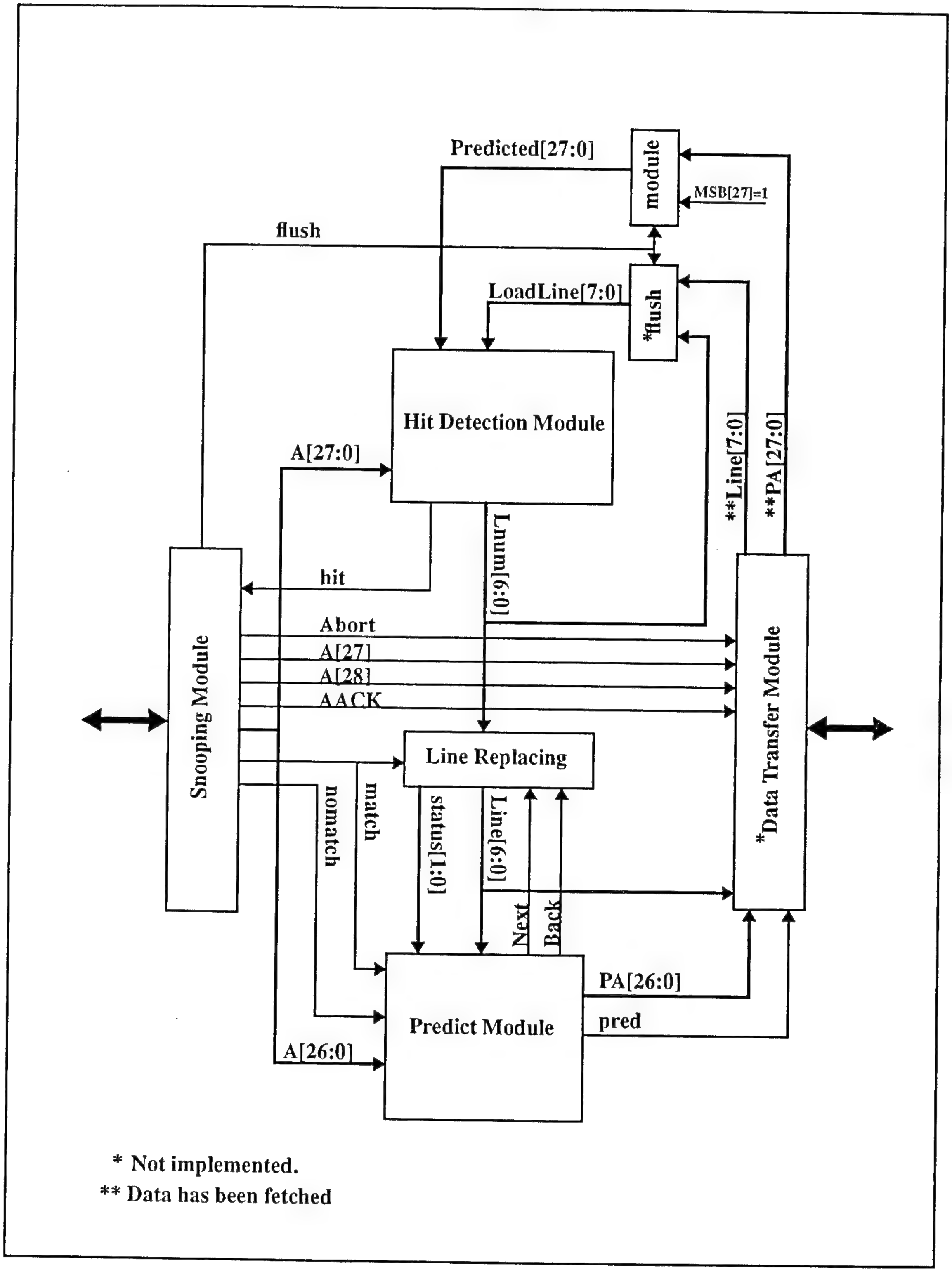



Figure 30: Intermodule Communication

# IV. IMPLEMENTATION OF PRC MODULES

## A. PROCEDURE

The modules described in the previous chapter were implemented utilizing Mentor Graphics' Design Architect, Cadence's Verilog-XL, and Cascade's Epoch 3.1. The approach taken was to first generate the design at the schematic level using Design Architect, and then describing the circuit either structurally or functionally using Verilog. The verilog description was then input into Epoch for automatic geometry creation in double metal, double polly, 1.2 micron technology.

## B. IMPLEMENTATION

The schematic sheets generated with Design Architect (DA) were built utilizing primitive and composite parts available from Epoch's library of parts. All sheets have been properly checked[1] and passed DA's tests with zero errors and zero warnings. Although these sheets could have directly been input into Epoch, it was decided to use them only as a visual guidance for the verilog programming. The decision was influenced by the ease of programming and fast simulation time provided by verilog. The files containing these sheets are located on line under the "../RPB.mgc" directory. A print-out of each of these sheets is included in Appendix A of this report.

The verilog files contain a structural description of the data path captured in the DA's schematic sheets. In addition, some files contains the functional description of the Finite State Machines for synthesis in Epoch at compile time. Other files are expressed as a combination of both descriptions. These files are contained in two directories. The "verilog" directory, contains the *verilog-in* files which are the ones that provide input to Epoch for geometry generation. The "vout" directory contains the *verilog-out* files which

1. Design Architects performs different types of checks on nets, instances, properties, frames etc.

are extractions of the generated geometries and contain delay information. The path to these directories is "../projects/PRC2/directory". A copy of the *verilog-in* file is included in Appendix B. The *verilog-out* file is not presented in this report. They are computer generated and are very extensive in size.

For each module, Epoch automatically created, placed, routed, and buffered the geometry. Table 2 presents the specifications provided to Epoch necessary for the generation process. Epoch outputs three types of transcripts; The netlist transcript, the placement, routing and buffering transcript, and the extraction transcripts. These transcripts are also extensive in size and a copy of them is not included in this report. They all are saved in the "transcript" subdirectory which is located at the same level of the "verilog" and "vout" directories. All Epoch files are contained in the "projects" directory.

The following sections provide additional information about the design and implementation of the completed PRC modules. Their simulations are given in the next chapter. See Appendix E for directions on how to view the generated geometries.

**Table 2: Specified Parameters for Module Generation**

| | |
|---|---|
| Design Rule | Orbit1.2u.2m.2p |
| Ambient Operation Temperature | 25 Celsius |
| Default Clock Frequency | 66.6 Mhz |
| Default Switching Factor | 50 % |
| Maximum Voltage Drop | 0.25 volts |
| Max. Simultaneous Switching Current | 40 mA |
| DC Current Limitation | 90 mA |

## 1. The Snoop Module

This module uses four instances of a parity checker, an instance of dff register and a Finite State Machine. Each instance of the parity checker generates an odd parity bit for an 8-bit address bus. The generated odd parity bit is logical 1 if there are an even number of 1's in the 8-bit address bus. This bit is compared against the odd parity bit provided by the cpu. If they are not equal an ERROR signal is asserted. The four error signals (one for each instance) are "OR" gated to determine the parity error signal which is sent to the finite state machine. Figure 31 presents the state diagram for this FSM. It fits nicely into a 3-

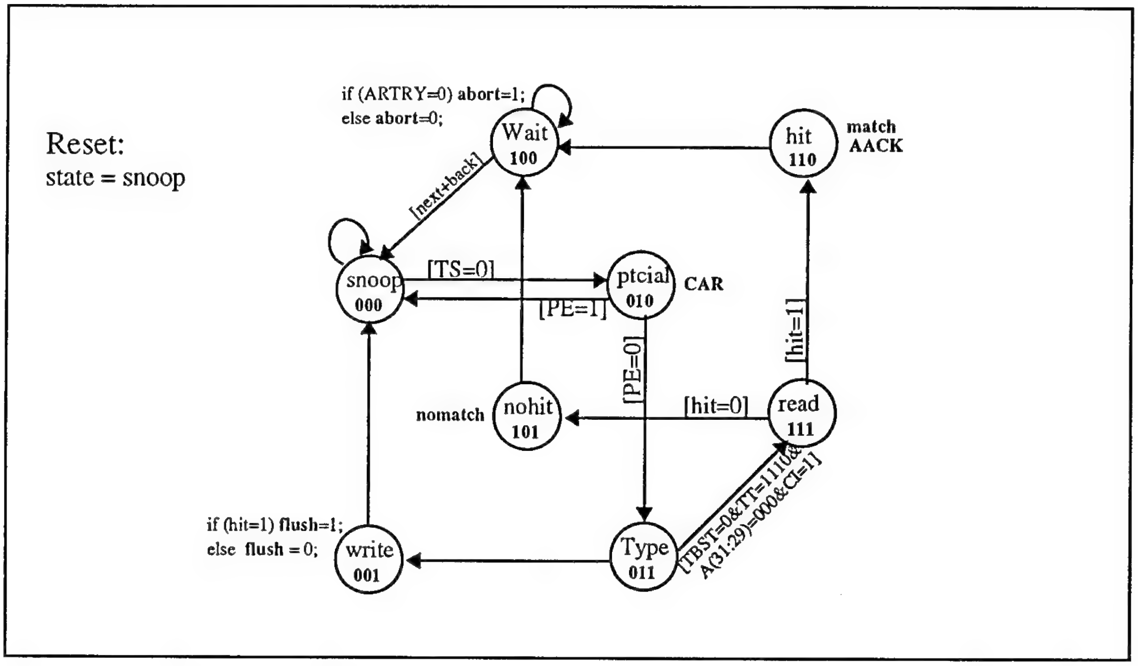



Figure 31: State Diagram for the Snoop Module FSM

dimensional hypercube. Notice that all communicating states are one Hamming distance apart, eliminating the well known hazard problem. Figure 32 presents the convention used for this state diagram and following ones. At initialization time, the register is clear and the FSM reset to the "snoop" state (000b) where it is ready to snoop for a potential access. When *TS* is asserted, the state changes to the "ptcial" state where the current address register gets loaded and the parity error signal is inspected. If an error exists the state moves

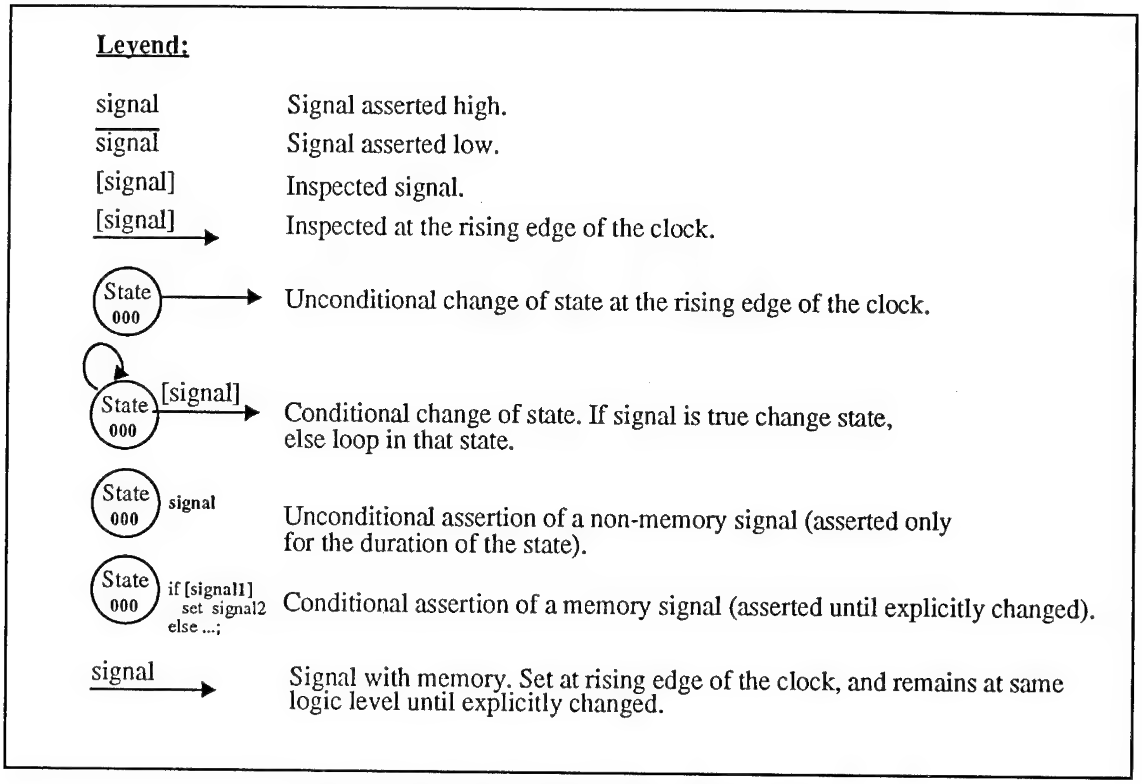



**Figure 32: Legend for State Diagrams**

back to the snoop position. Otherwise, it passes to the "type" state where the appropriate signals are inspected to determine whether the access is a read or a write operation. For a write access, the *hit* signal determines whether or not the *flush* signal is set. Similarly, for a read operation, the *hit* signal determines which state is entered. For this signal to correctly be inspected, it needs to be provided within 3 clock cycles after the register is loaded. The hit detection module provides the signal in about 2 clock cycles when the clock is configured to a period of 15 nanoseconds (66.6 Mhz). Because the parity checker basically constitutes the data path of this module, only this unit is shown in the schematic sheets presented in Appendix A. The verilog description includes both the structural description of the data path and the functional description of the FSM. The module was implemented with what Epoch calls "Standard Cell" design, in which the physical module is optimized for pitch-matched row placement rather than placement in a regular, bus-oriented grid.

## 2. The Hit Detection Module

This module is composed of two sub-modules; the *hitmod* module and the *encoder* module. The first sub-module contains the set of registers and comparators as well as the decoder and the bank of buffers. The second sub-module implements the priority encoder, which is one of the few parts not available in epoch's library. There is no strong reason for keeping both as separate modules. Perhaps the only motivation is that *hitmod* is the biggest module of the project and requires a great deal of time and system memory to compile and simulate. Therefore, keeping the encoder out expedites the process of compiling and simulating. The next sections provides information concerning the design and implementation of each of these two module.

### a. *The hitmod Module*

The *hitmod* module contains an instance of a decoder, a set of buffers, and eight instances of the pbank component. The decoder is basically a PLA module without the OR plane. It is synthesized at epoch compile time and its operation is expressed in a tabular code file similar to a truth table. The decoder8x128 codefile is presented in Appendix B. The set of buffers are used to aid the instances and handle fanout. Some are implemented as standard cell and others as buscell (vectored bitwidth). The pbank (prediction bank) component is a module by itself. It has been compiled and simulated. This hierarchical structure significantly reduced the placement, routing and buffering time of the hitmod module. The pbank module implements sixteen prediction lines. Figure 33 shows the circuit diagram for one of the lines. The 28-bits of the predicted address are separated into an upper and lower nibble to store them in two separate registers. At initialization time, the lower nibble is cleared and the upper nibble is preset (setting the flag bit to logical 1). This configuration is the product of several different attempts. With just one 28-bit wide register, the clock line could not handle the fanout. In addition, it was found that each line worked better when implemented as a standard cell vice datapath cell (when implemented as a datapath, Epoch arrange the cells in a bus-oriented, row-and-column

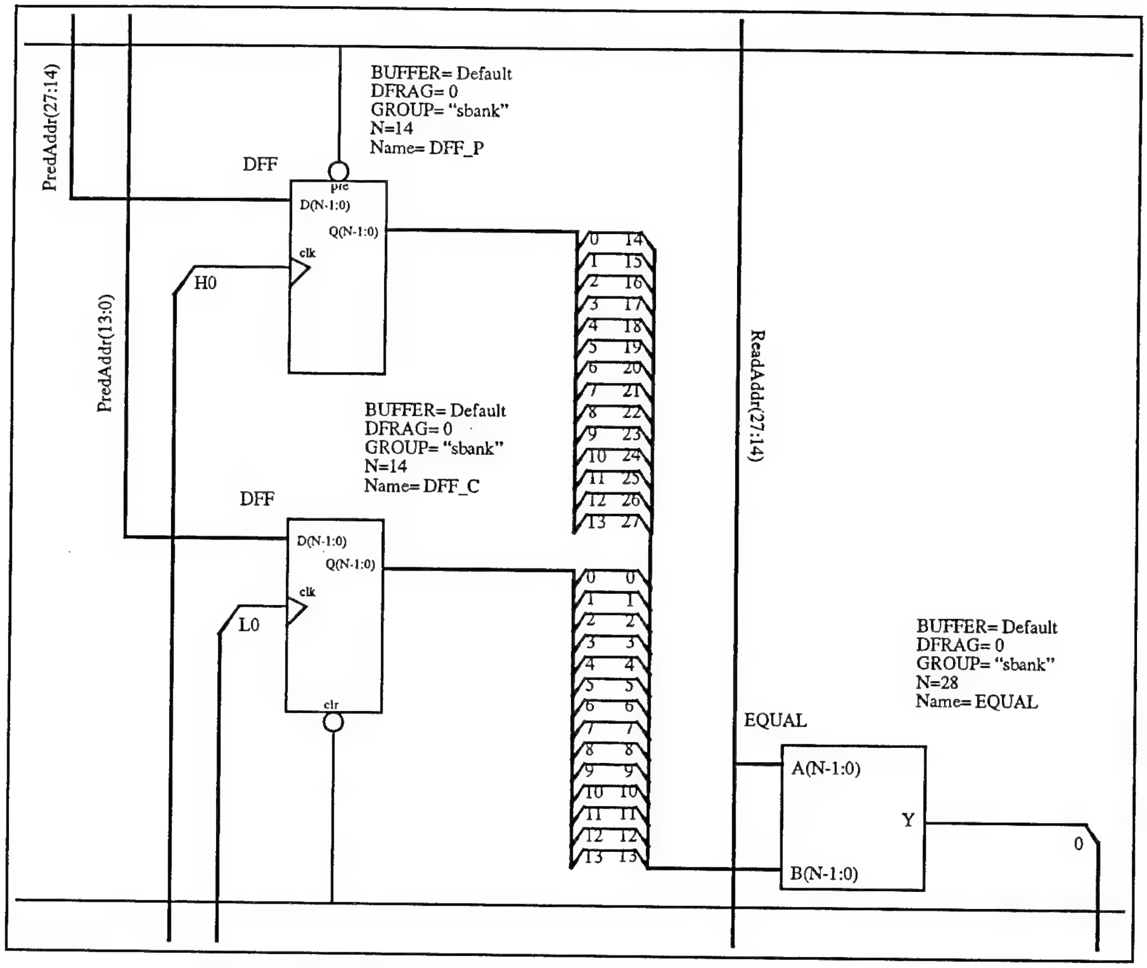



**Figure 33: Single Prediction Line Circuit Diagram**

architecture. Datapath is typically more area-efficient than an equivalent set of standard cells and has more balanced timing characteristics across its bitwidth). Finally, the module was given the fixed block attribute, which means that it will remain as a block in the next higher level design and will not be smashed or absorbed by that design. This is done to preserve the electrical characteristics of the module and to reduce the compiling time during placement and routing.

### *b. The encoder Module*

This module was designed using the same technique utilized for the design of the commercially available 74LS148, MSI 8-input priority encoder [Ref. 12]. The following explains this technique and the way it was extended to encode 128 lines:

The first eight intermediate variables are defined (H0-H7). The variables are then prioritize with respect to the inputs, according to the following logical equations[2]:

$$H7 = I7 \quad \text{(Eq 4.1)}$$

$$H6 = I6 \,.\, {\sim}I7 \quad \text{(Eq 4.2)}$$

$$H5 = I5 \,.\, {\sim}I6 \,.\, {\sim}I7 \quad \text{(Eq 4.3)}$$

$$\ldots \qquad \ldots$$

$$H0 = I0 \,.\, {\sim}I1 \,.\, {\sim}I2 \,.\, {\sim}I3 \,.\, {\sim}I4 \,.\, {\sim}I5 \,.\, {\sim}I6 \,.\, {\sim}I7 \quad \text{(Eq 4.8)}$$

Table 2 presents the Truth table for a simple binary 8-input encoder. From the table, equations 4.9, 4.10 and 4.11 are obtained.

**Table 3: Truth Table for an 8-bit Encoder**

| Inputs | Outputs | | |
|---|---|---|---|
| Temp. var. | A2 | A1 | A0 |
| H0 | 0 | 0 | 0 |
| H1 | 0 | 0 | 1 |
| H2 | 0 | 1 | 0 |
| H3 | 0 | 1 | 1 |
| H4 | 1 | 0 | 0 |
| H5 | 1 | 0 | 1 |
| H6 | 1 | 1 | 0 |
| H7 | 1 | 1 | 1 |

$$A2 = H4 + H5 + H6 + H7 \quad \text{(Eq 4.9)}$$

$$A1 = H2 + H3 + H6 + H7 \quad \text{(Eq 4.10)}$$

$$A0 = H1 + H3 + H5 + H7 \quad \text{(Eq 4.11)}$$

2. The symbols ~, . and + are used as boolean operators.

The output equations of the priority encoder are obtained by substituting equation 4.1 through 4.8 into the appropriate equations, Eqs 4.9-4.11. Manipulating the results and performing boolean simplification, the following output equations are obtained:

$$A2 = I4 + I5 + I6 + I7 \qquad \text{(Eq 4.12)}$$

$$A1 = (I2 \,.\, \sim I4 \,.\, \sim I5) + (I3 \,.\, \sim I4 \,.\, \sim I5) + I6 + I7 \qquad \text{(Eq 4.13)}$$

$$A0 = (I1 \,.\, \sim I2 \,.\, \sim I4 \,.\, \sim I6) + (I3 \,.\, \sim I4 \,.\, \sim I6) + (I5 \,.\, \sim I6) + I7 \qquad \text{(Eq 4.14)}$$

One additional output is obtained by Eq 4.15. This output can be looked as "Got Something". The "GS" signal is asserted if any of the inputs is asserted. This is particularly useful for the creation of bigger encoders.

$$GS = I0 + I1 + I2 + I3 + I4 + I5 + I6 + I7 \qquad \text{(Eq 4.15)}$$

The prioritizing and encoding of 128 lines was done in the following manner; A module was created to prioritize 64 lines. To implement this 64 line encoder, eight instances of an 8-input priority encoder were used. Each encoder instance simultaneously applied Eqs 4.12 through 4.15 to eight different lines. The resulting eight A2-A0 outputs were tri-stated and connected together. This determines the lower 3 bits of the final output. Equations 4.12 through 4.15 were again applied to the eight resulting "GS" outputs. This prioritizes and encodes the eight groups. Now $A2_2$-$A0_2$ determines the upper 3 bits of the final output. The "$GS_2$" was combined with the "$GS_2$" of a second instance to form the 128 priority encoder. The logical "OR" of the two "$GS_2$" determines the *hit* signal.

A hierarchical structure was also used to construct the *encoder* module However, the encoder8x3 and the encoder64x6 sub-modules were given a non-fixed block attribute. Hence, they were smashed and absorbed at the highest level. Also, because of the nature of these modules, their implementation was specified to be in a standard cell form.

### 3. The Predict Module

The data path of this module was implemented using two instances of a register file, one instance of an adder, and various instances of a buffer, inverter, and "AND" gate. The register file has one write and one read port and is capable of storing 64 lines. It is available only in datapath form. The set of buffers and gates are used to decode the most significant bit of the write/read address in order to select the appropriate register file instance. The inverting buffer provides a logically inverted input to the register file. The adder is a carry look ahead adder with carry-in bit, also implemented in datapath form. The finite state machine was minimized and synthesized at compile time by Epoch. Figure 34 presents the State diagram for this FSM. The states, fit nicely into a four dimension hypercube. Each cube keeps track of a "nomatch" signal. If a match occurs in between the two consecutive nomatches, the states cycle through the prediction process and return to

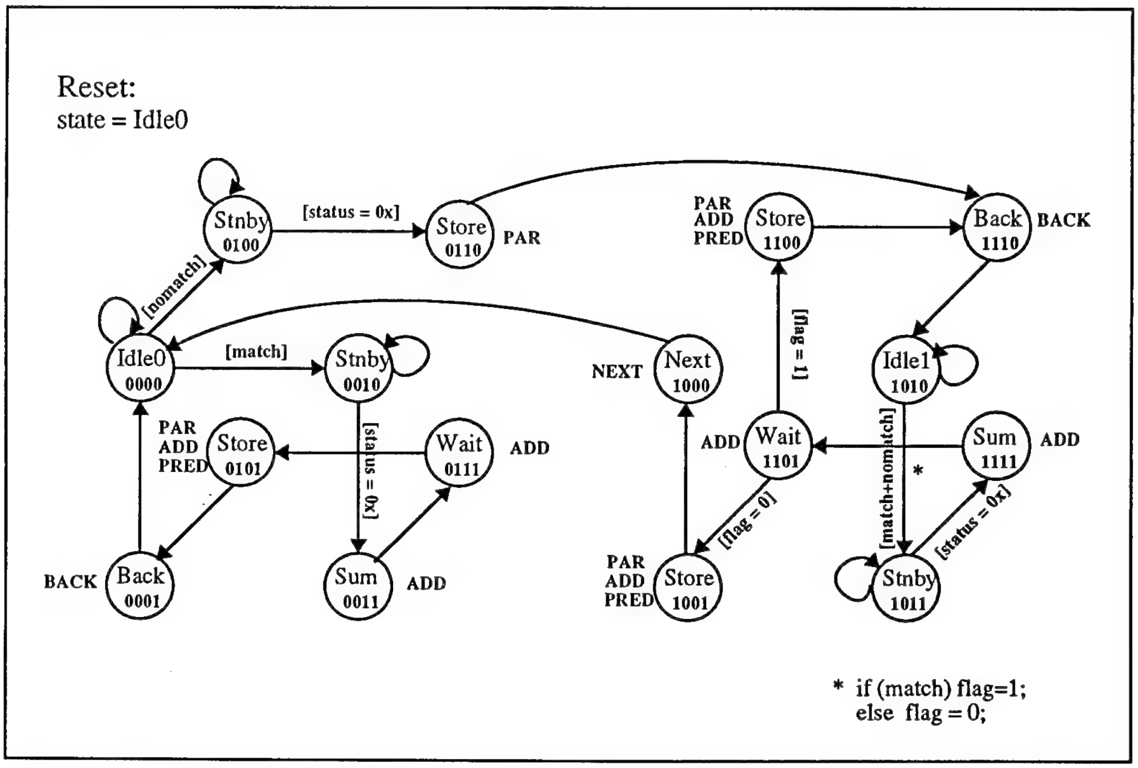



**Figure 34: State Diagram for the Predict Module FSM**

the place where it was disturbed. At initialization time, the machine is reset to the Idle0 state. In this state, the *match* and *nomatch* signals, asserted or negated by the snoop module, are inspected at the rising edge of the clock. If a nomatch is received, the state changes to the standby state where the line status signal is inspected. This state waits until the line number is valid (if the replacing module is working properly, this signal should already be set to valid and the process is delayed by only one clock cycle). The FSM proceeds to issue a *PAR* signal (store address as previous in register file) and *Back* signal (hold the current line number) and then settles in the idle1 (1010b) state where it again inspects the *match* and *nomatch* signals. In this state, if either one is asserted, the FSM records in a flag which signal was the asserted one and then proceeds to perform a predict cycle[3]. The FSM inspects the flag bit in the Wait state (1101b) to determine whether it should reset to the idle1 or idle0 position. If it is to the Idle0, the machine sends a *next* signal indicating that a new line number should be provided. Otherwise, it sends a *Back* signal to restore the line number displayed prior to the hit.

### 4. The Line Replacing Module

Figure 35 presents the data path for this module. The binary *mod* 128 counter is implemented by a high speed ROM and a D-register. The ROM operation is best described by the following equation:

$$\text{Output} = \begin{cases} K+1 & \text{for } 0 <= K < 126 \\ 0 & \text{for } K=127 \end{cases} \qquad \text{(Eq 4.16)}$$

where K is the binary input to the ROM. The equation expresses that the ROM holds the value (K+1) at location K (for; 0 <= K < 126) and the value (K-127) at location K=127. When an increment is required, the ROM's output is fed back to it's input by loading the feedback register. The ROM operation is specified in a codefile (presented in Appendix B) and is synthesized at epoch compile time. The flags are set and cleared in a

3. The reader is reminded that both signals *match* and *nomatch* are asserted for one clock cycle. Therefore, the FSM needs to save which one occurred for later use.

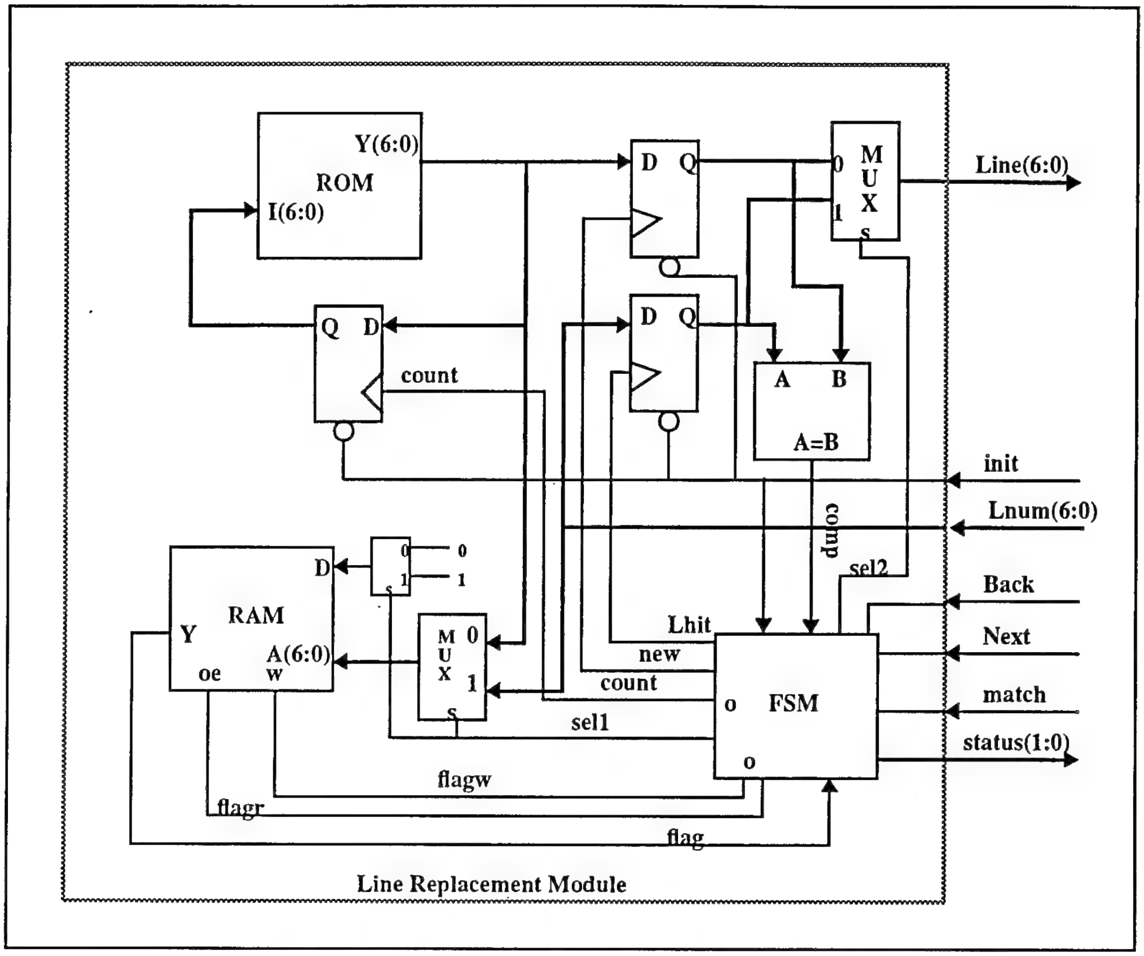



**Figure 35: Data Path of the Line Replacing Module**

static RAM module which has static address decoding for fast access time and reduced power requirements. The FSM for this module also fits nicely in a four dimensional hypercube. Figure 36 presents the finite state machine for this module. At initialization time, all the registers are cleared and the state machine is reset to the ready state. In this state, the zero input is selected in all the muxes and a status of "valid and ready" is displayed (the line number is valid because the register's output is 0d, hence line number zero is ready to be used). The feedback register also feeds 0d to the ROM, making the ROM's output to be 1d. At this point the flag is not inspected because no line has yet been used. No further action takes place and the FSM loops in the ready state until either a *next*

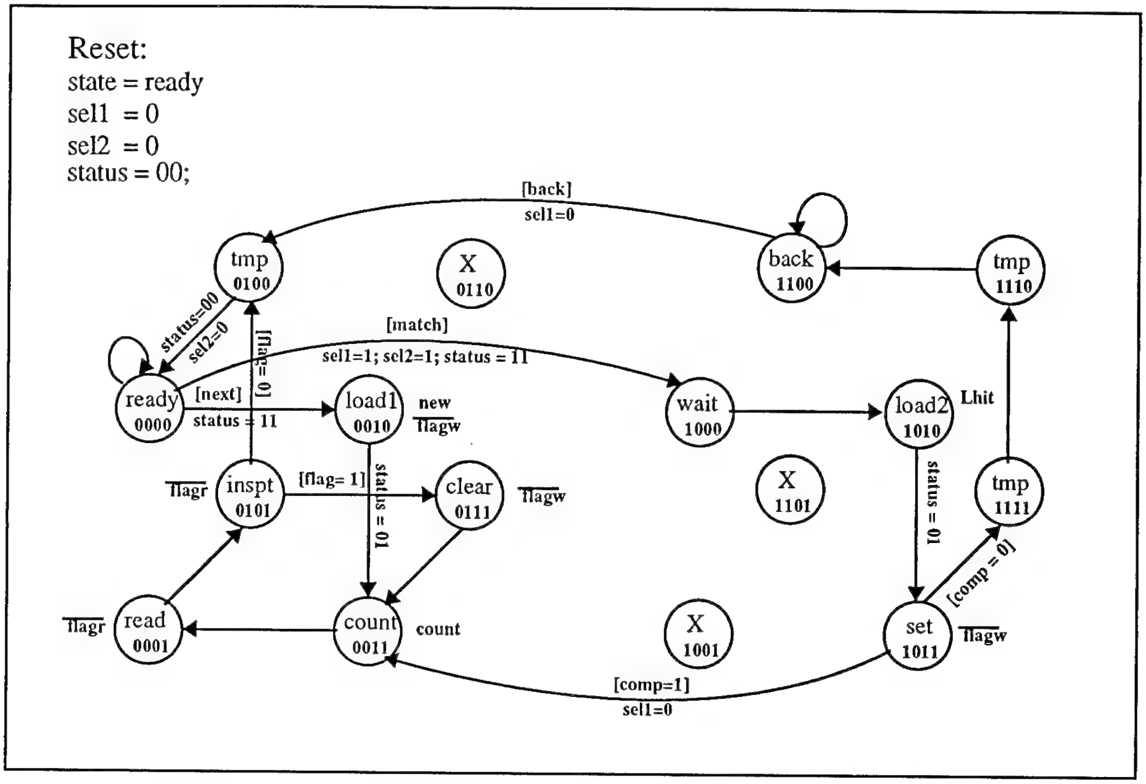



**Figure 36: State Diagram for the Line Replace Module FSM**

or a *match* signal is received. When either one of these signals is received, the FSM performs the operation as explained in the previous chapter. It is important to point out certain aspects of the design. This module determines the maximum clock rate of the system. For the module to correctly inspect the flag signal, the clock pulse between the "count" state (0011b) and the "read" state (0001b) must be higher than the time it takes to increment plus the time it takes to mux and hold the increment. This is best shown by equation 4.17.

$$T_p > (T_{dff} + T_{rom} + T_{mux} + T_{setup}) \qquad \text{(Eq 4.17)}$$

where,

$T_p$ is the minimum clock period
$T_{dff}$ is the delay time from the clock input to the data output of the dff-register.
$T_{rom}$ is the delay from input to output of the rom
$T_{mux}$ is the delay from input to output of the multiplexer
$T_{setup}$ is the address setup time of the RAM.

# V. SIMULATIONS

## A. GENERAL

Once a verilog description was obtained for a module, the code was simulated at the logical level before being input into Epoch (*verilog-in* files). After geometry creation, an extraction was performed and output in verilog form for a second simulation (*verilog-out* files) which included capacitance load and delay information. Both simulations were done utilizing the same testshell file. These files are presented in Appendix C. Two copies of each of the simulation files exist. One copy is located in the verilog directory and the other copy in the vout directory. In both directories, the *verilog-in/ verilog-out* files have a ".v" extension and the testshell files have a ".i.v" extension. In addition, the vout directory contains files with a ".sdf" extensions. These files provide the capacitance load and delay information for the respective ".v" file. Furthermore, both directories contain files with ".zsim" extensions which are the transcripts obtained from the simulations. Because of the large size of these transcripts, a copy is not provided in this report. Verilog also outputs a time waveform diagram which can not be saved or nicely captured due to its black background. To simulate a *verilog-in* file, the following command should be used (in the verilog directory):

"prompt> verilog +libext+.v+ -y /tmp_mnt/local/epoch/models/cmos/verilog filename.i.v filename.v"

the ".v" file should be followed by any other ".v" files which are part of the hierarchy. The *verilog-out* files are simulated by using the following command (in the vout directory):

"prompt> verilog -v /tmp_mnt/local/epoch/data/verilog/primlib.v filename.i.v filename.v"

this time the ".v" file is not followed by any other file, not even the ".sdf" file (verilog automatically looks for it in the current directory).

The simulations performed are far from being exhaustive, nevertheless they cover and test the basic and fundamental parts of each module. The following sections present a brief

description of the simulations performed in each of the implemented modules. The description is intended to provide enough information for the understanding and editing of the created test.

## B. THE SNOOP MODULE

The snoop.i.v testshell file tests both the data path and the finite state machine of this module: The program starts by initializing the FSM and interface signals, then proceeds to simulate five accesses. The first one is a write access with wrong parity bits. The parity checker of the module detects this and sends a parity error to the FSM. During this access, the FSM enters state 2 and returns to state 0 upon receiving the parity error signal. The second access is a correct write access with no hit. The FSM successfully cycles through the 0-2-3-1-0 states and does not raise the *flush* signal. The third access is also a correct write access, but this time with a hit. The FSM cycles again through the same states and raises the *flush* signal. The fourth access is a correct read with no hit. This forces the FSM to cycle through the 0-2-3-7-5-4-0 states. The last access is a correct read with hit. The FSM follows the 0-2-3-7-6-4-0 path. This completes the test for the module.

## C. THE HIT DETECTION MODULE

Three testshell files exist for this module. The hitmod.i.v, the pbank.i.v, and the encoder.i.v file. The first two files are basically the same, the first one tests for the entire set of 128 lines and the second one for a subset of 16 lines. Only the pbank.i.v file is explained here, because the rational is the same for the other one. The encoder.i.v tests the encoder submodule.

The pbank.i.v testshell program starts by initializing all the registers. The upper nibble register (bits 27-14) is preset and the lower nibble register is cleared. Two nanoseconds later, a requested address (0000002h) is input. All the comparators find a no match condition because no predicted address has been saved (MSB is 1). Later a predicted address 0000003h is stored in line 0. Four nanoseconds later, the requested address changes to 0000003h and a match condition is found by the comparator of line 0. The same process

is repeated for line 16, then for line 1. This completes the program. There is no real need to verify every single line because they all are replicas of line 0.

The testing of the priority encoder is simple. A number is presented at the 128-bit bus input. If the number is anything but zero, the encoder should assert its hit output and provide the binary number of the position of the most significant 1 of the input. Notice that if a match is found in line 0, the input to the encoder will be...001*h*. The program encoder.i.v first presents the number 00000000000000000000000000000000*h* and the encoder outputs hit=0. Ten ns later, it presents 0000000000000000ffffffffffffffff*h*. The encoder outputs hit=1 and Lnum=3f*h* (decimal 63), which is correct because the most significant 1 is in position 63. An easy way to verify this is to multiply the number of f's by 4 and subtract 1. The program next presents number 000000000000000fffffffffffffffff*h* for a Lnum output of 43*h*. Later, it presents 000000000fffffffffffffffffffffff*h* and then ffffffffffffffffffffffffffffffff*h* (which is the worst case). The output of the encoder is 5b*h* and 7f*h* respectively. This terminates the testing of this module.

Something not mentioned about the hit detection module is that both submodules were not assembled together in a higher hierarchy. As explained, they were separated because of compiling and simulation latency. Introducing an additional layer would not be beneficial at all. The best way is to glue them at the top-most hierarchy where all the modules are put together.

### D. THE PREDICT MODULE

The data path and Finite State Machine of this module were also implemented and simulated independently. Similarly to the hitmod module, the register files and adder of this module constitutes a big portion of the chip and they take a great deal of cpu time and system memory to simulate. The predict.i.v program tests the data path of this module. The program presents seven consecutive read accesses. The first address 0000001h is latched into the previous address register of line 0. Like the pbank module, there is no need to switch and test every single line number, therefore the register file is fixed to line 0. When

the second address of 0000002h is presented, a predicted address of 0000003h is successfully generated by the adder. The second address now overrides the first address. A new address of 0000003h is presented and the process repeats, generating 0000004h. A fourth address of 000005h is presented and a predicted address of 0000007 is generated. The interesting part comes when the address of 0000000h is presented. The displacement is -0000005h. The adder produces the address 7ffffb*h*, which is also correct. The address 7ffffff*h* is the maximum address addressable by A(26:0). This proves the point that the developed technique wraps around the ends. To confirm that it also works the other way around, the programs shifts to line 1 and stores the address 7ffffd*h*. Then it presents address 7ffffff*h*. This gives a displacement of 0000002h, which added to the presented address, gives 10000001h. The module throws out the MSB and outputs 0000001h, which is the correct prediction.

The pmfsm.i.v program was created to test the finite state machine that controls the data path of this module. The idea is to create cases and force the FSM to enter all of its states. The program starts by resetting the sate machine to state 0. Then, it simulates a *match* case which makes the machine cycle through states 0-2-3-7-5-1-0. It proceeds with a *nomatch* case. The FSM moves from the idle0 position to the idle1 position along the 0-4-6-e-a path. Another *match* is issue and the FSM cycles again. This time, through a-b-f-d-c-e-a states. The program terminates with a *nomatch* case to return to the idle0 position along the b-f-d-9-8-0 path.

## E. THE LINE REPLACING MODULE

The linerep.i.v program tests both the data path and the Finite State Machine of this module. The procedure is similar to the previous module. The FSM is forced to enter all of its states. At initialization time, the FSM is reset to state 0. At this time, line number 0 is displayed. A *match* at Line number 4 case is simulated. The FSM replaces line number 0 with line number 4 for a period of time and then restores line number 0 when a *back* signal is received. For this, the machine cycles through states 0-8-a-b-f-e-c-4-0. This causes the

flag of line number 4 to be set. The program proceeds and issues 6 consecutive *next* signals. This forces the FSM to cycle 6 times through the rest of the states (0-2-3-1-5-4-0). At each *next* signal, the line number should increment. When the fourth *next* signal is received, the line number jumps from 3 to 5 since line number 4 is skipped because its flag was set when inspected at state 5 (this is the only time that state 7 is entered). This completes the program and the simulation process.

# VI. CONCLUSIONS AND RECOMMENDATIONS

## A. CONCLUSIONS

The results show that the Read Prediction Buffer IC works correctly and fully implements the intended algorithm. This is a positive sign for the VLSI design process utilized at the Naval Postgraduate School. Furthermore, the testing of the RPB gave insight into the design of the Prediction Read Cache. This enhanced version was not completely designed and implemented as part of this thesis work. However, the basic structure and ideas have been documented, and they will be of great significance to future research. Four out of six modules were designed and implemented. These modules were successfully simulated in isolation, but they may require some synchronization patching when glued together in the next higher hierarchy. Finally, this document may serve as a guide when the testing of the finished PRC comes about, since the process of testing an IC with nontrivial algorithms is not well explored in the literature.

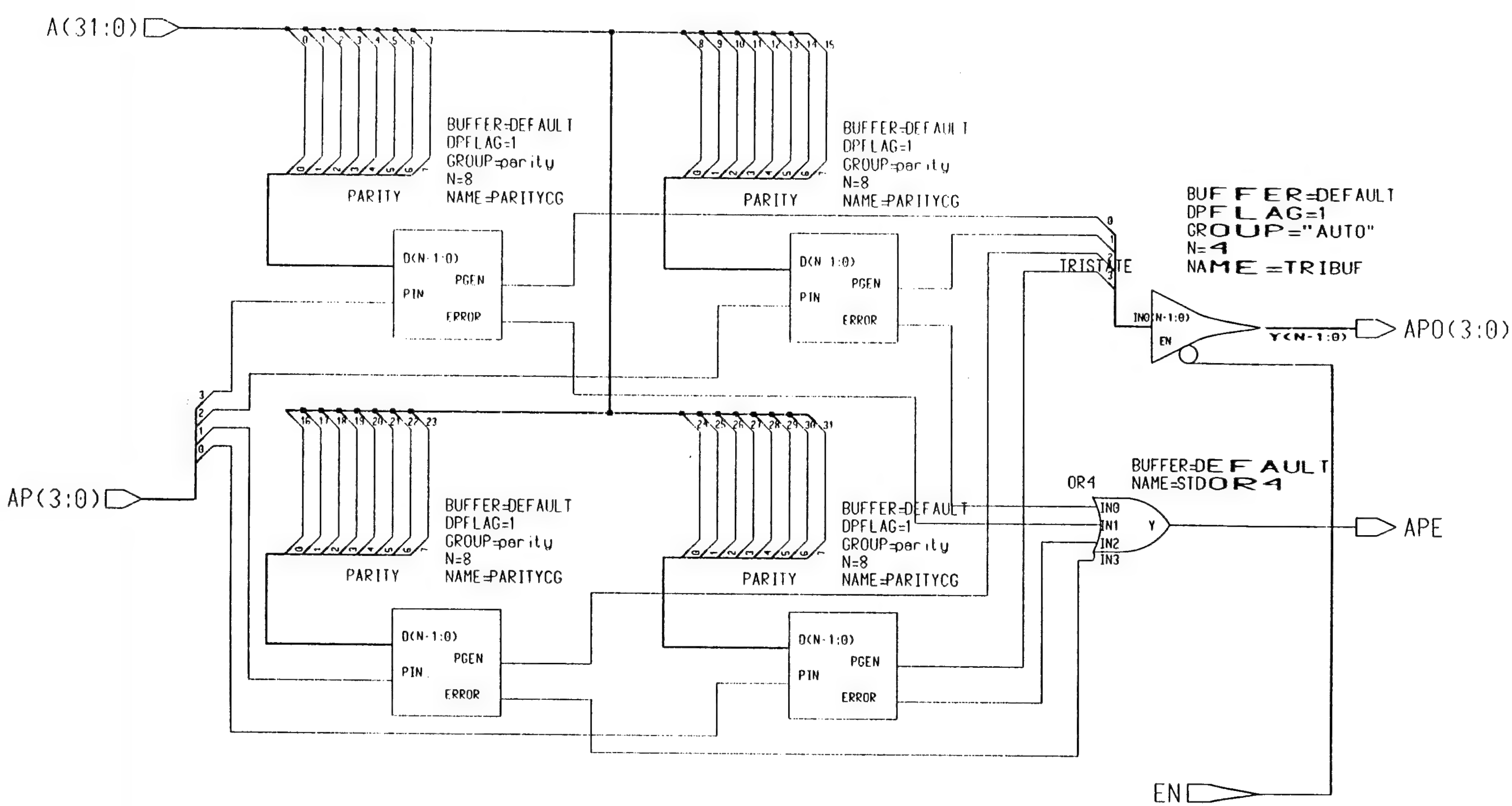
## B. RECOMMENDATIONS

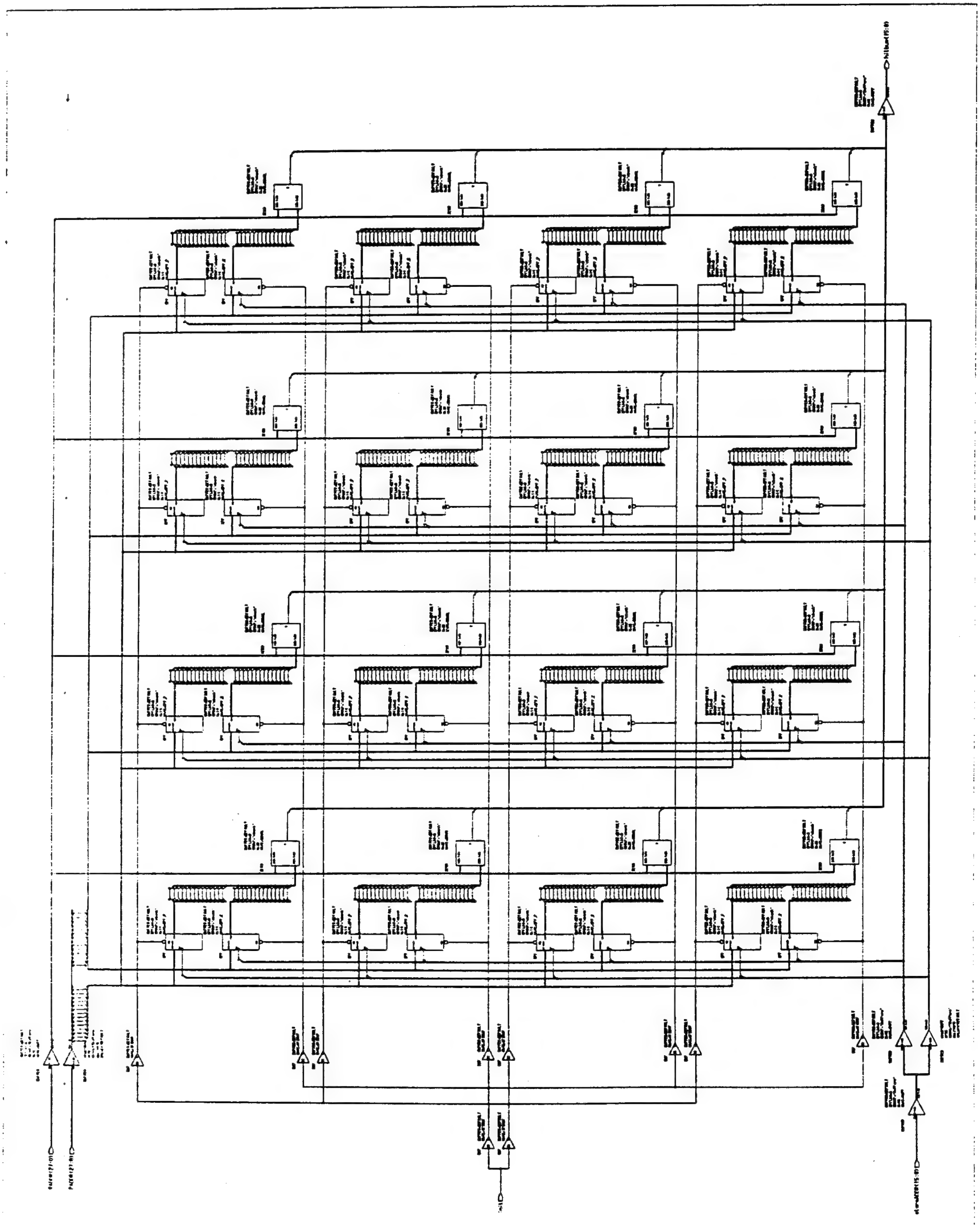
The algorithm implemented by the RPB and PRC integrated circuits embodies a novel approach to improve memory subsystem performance. It is recommended to future system developers to consider the presented approach and seek further gains in performance from improvements to the memory hierarchy, at least until significant advances in DRAM IC access times are made.

# APPENDIX A. SCHEMATIC SHEETS

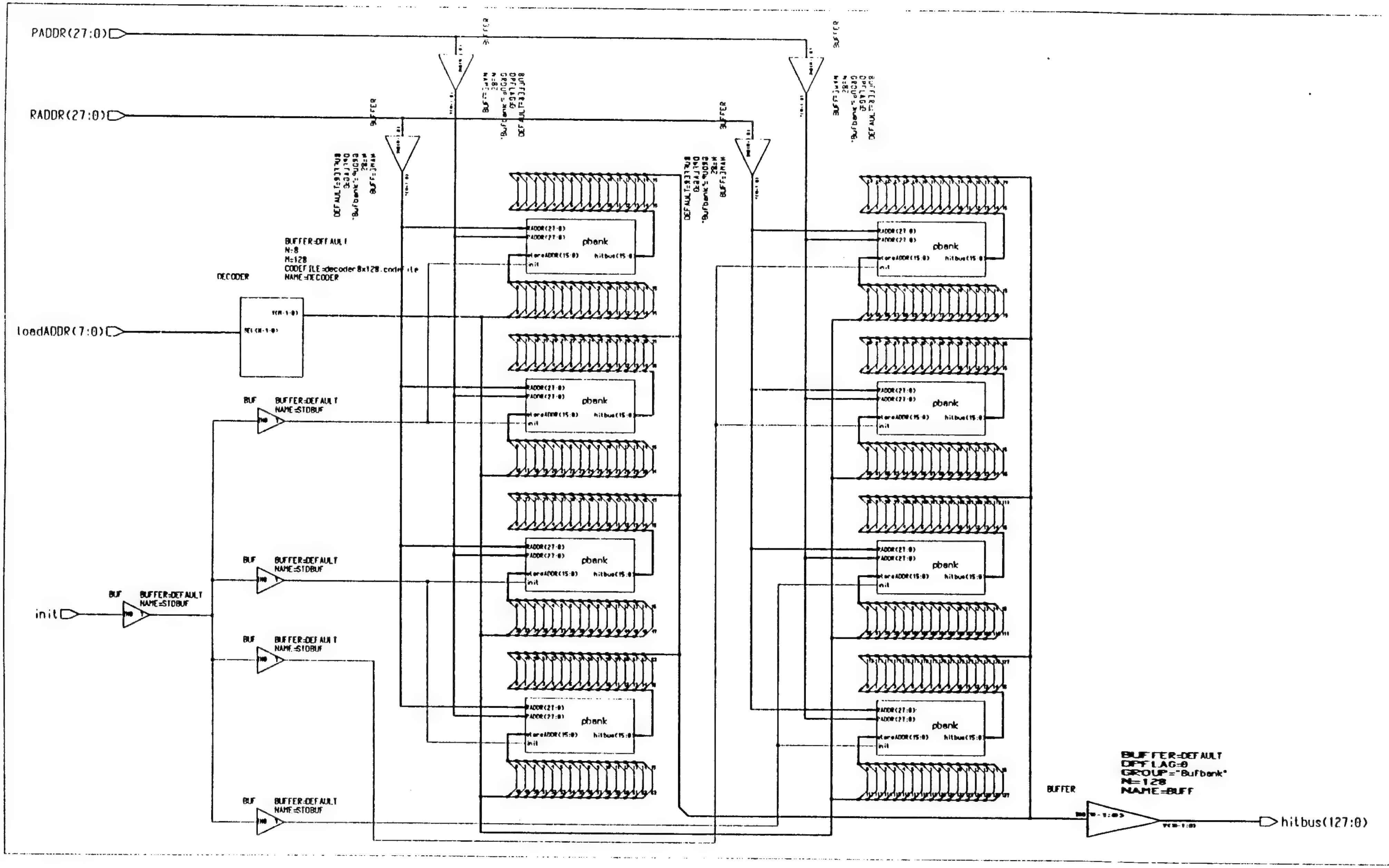
This appendix contains the schematic sheets for the data path of all the implemented modules of the Predicted Read Cache IC.

# Parity Checker for The Snoop Module

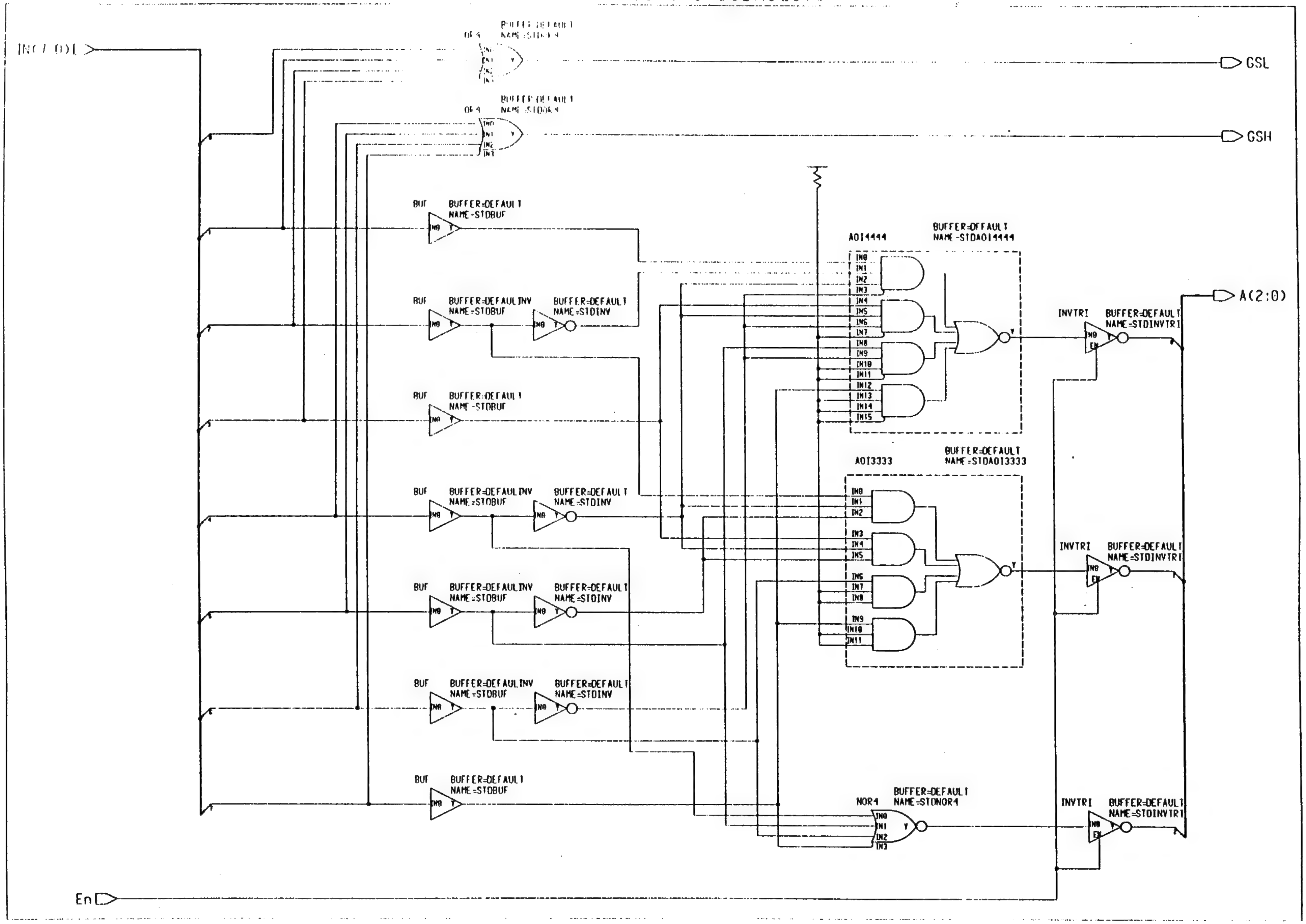




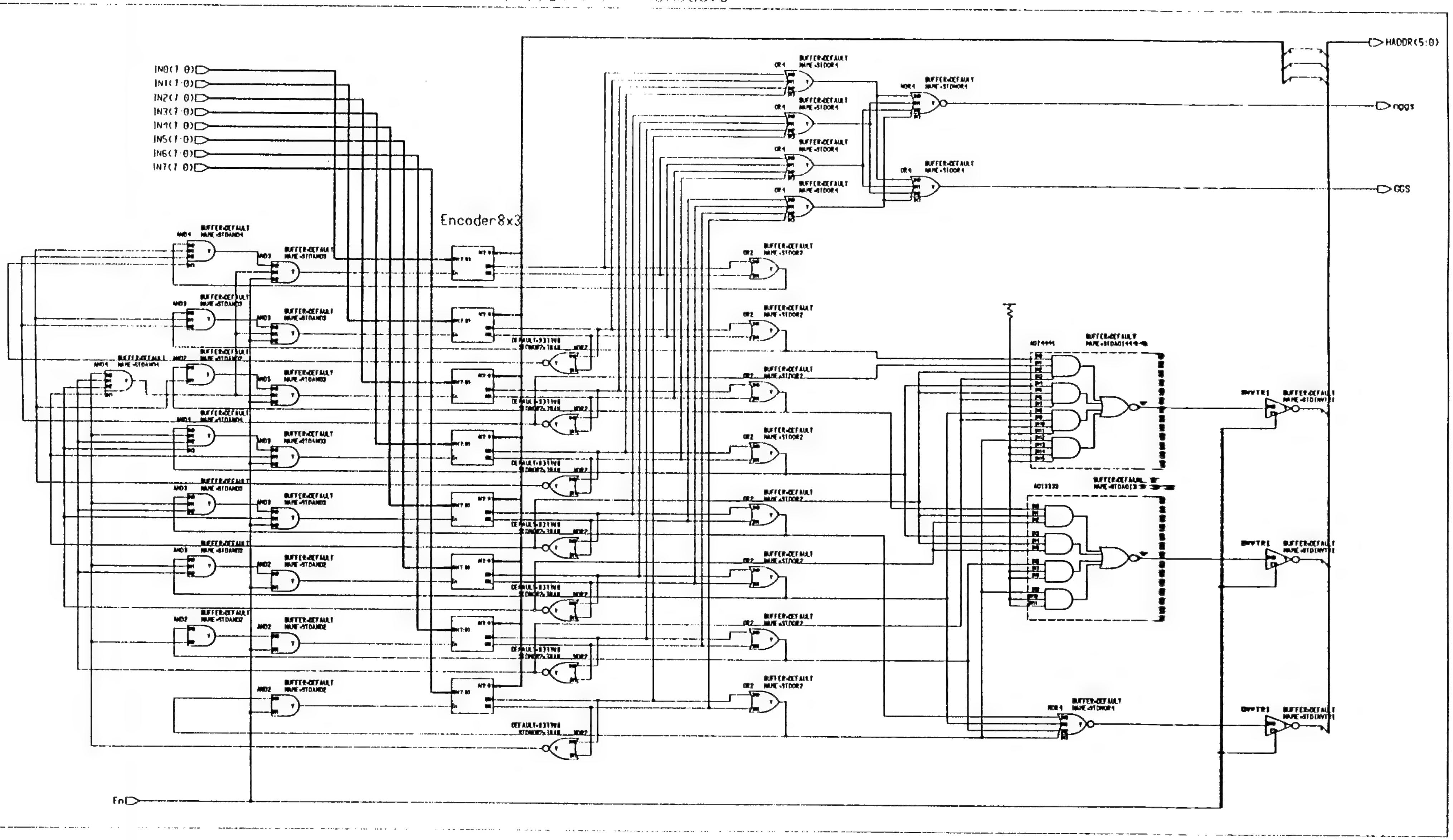
Hitmod module

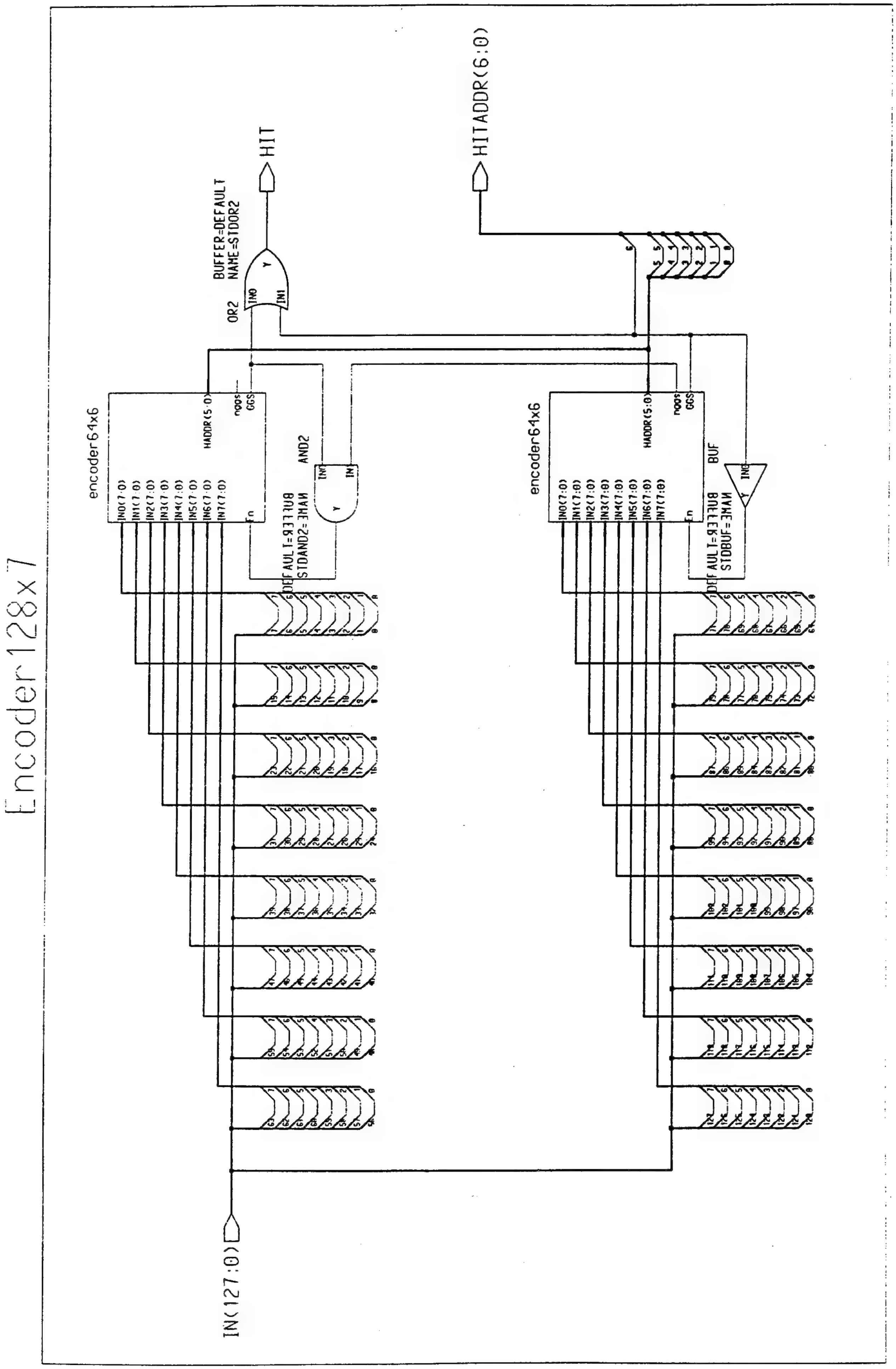


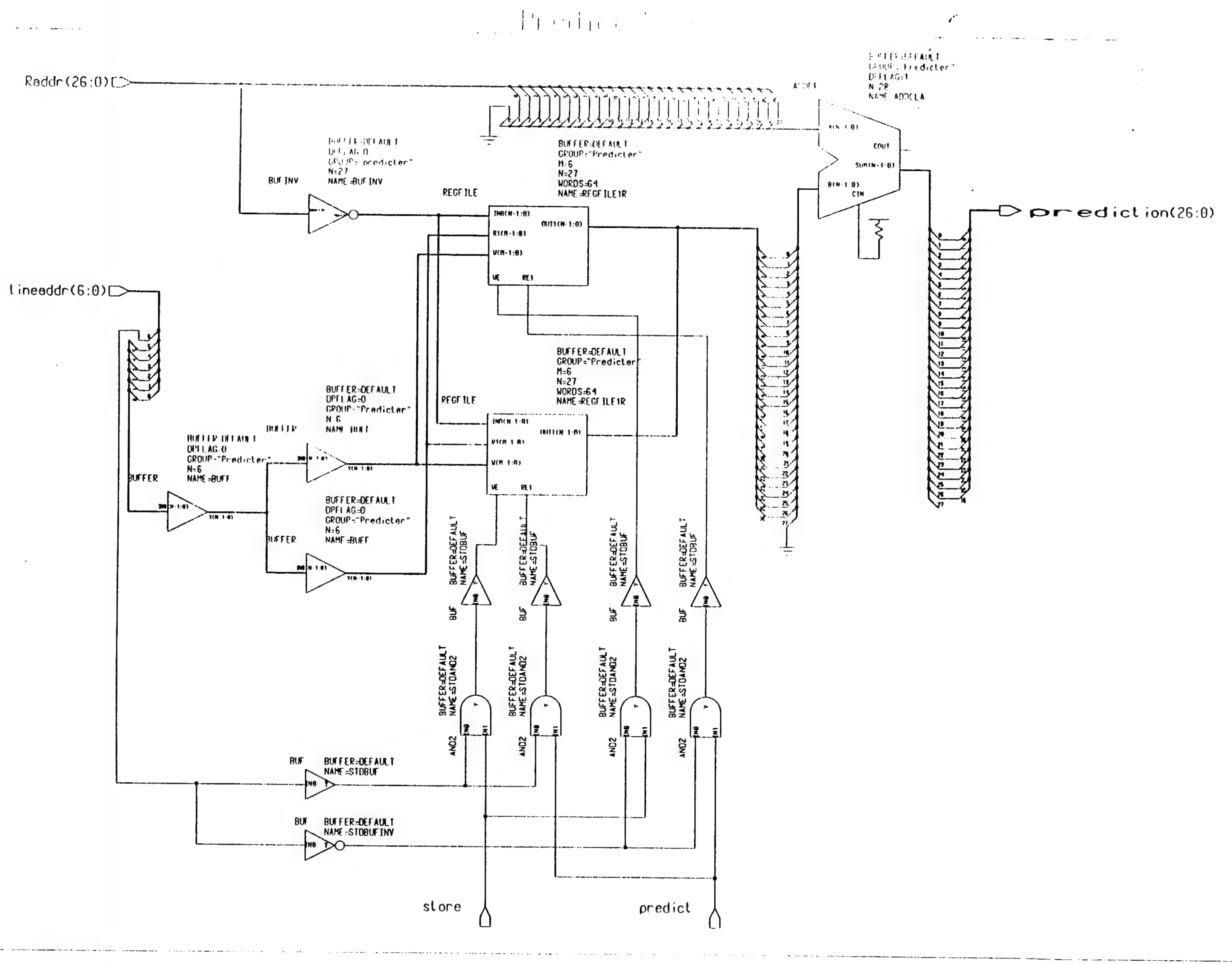
# Encoder8x3 Submodule



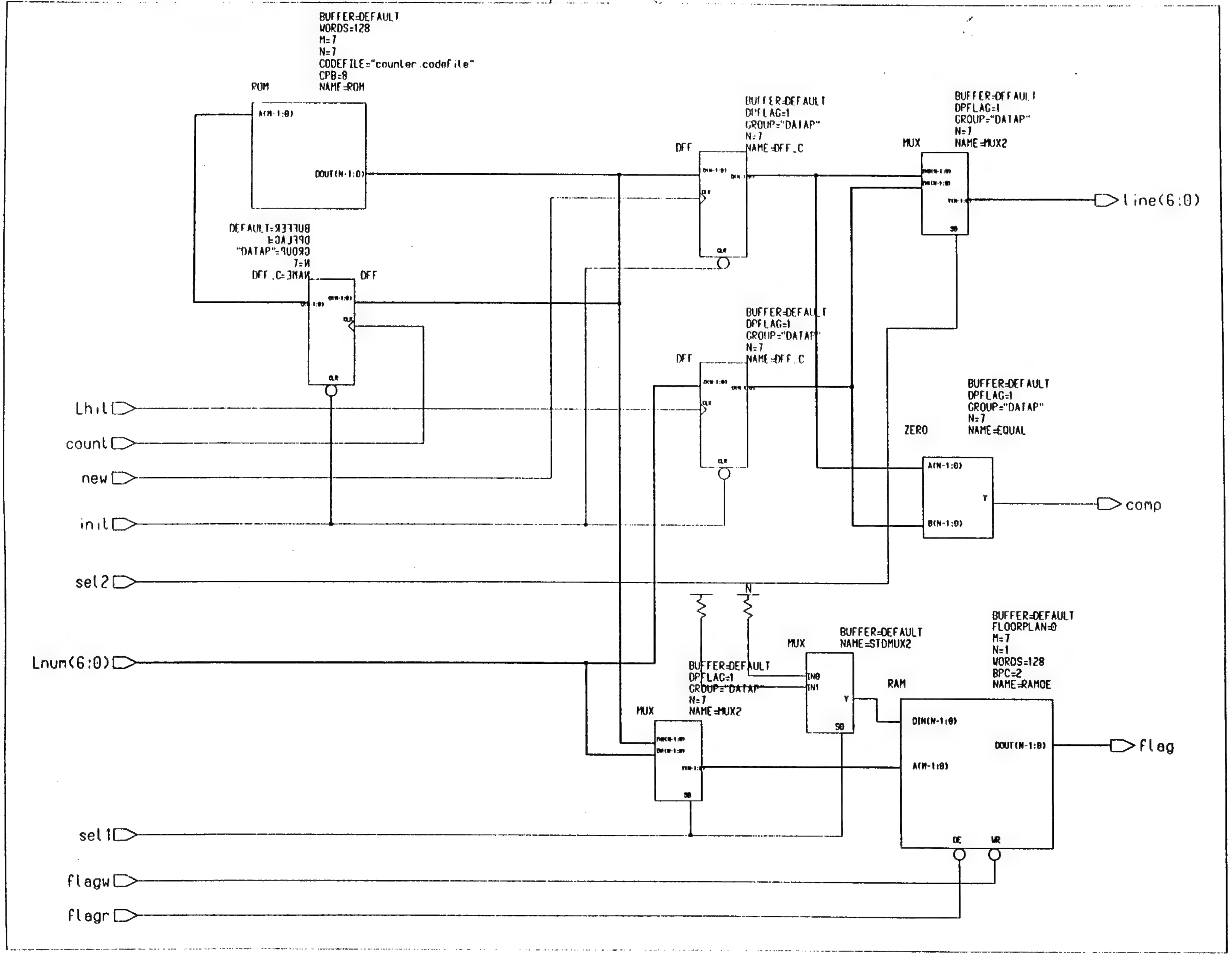
Encoder64x6 Submodule




Encoder128x7
IN(127:0)
HIT
HITADDR(6:0)
BUFFER=DEFAULT
NAME=STDOR2
OR2
encoder64x6
IN0(7:0)
IN1(7:0)
IN2(7:0)
IN3(7:0)
IN4(7:0)
IN5(7:0)
IN6(7:0)
IN7(7:0)
HADDR(5:0)
nGGS
GGS
En
AND2
BUFFER=DEFAULT
NAME=STDAND2
encoder64x6
IN0(7:0)
IN1(7:0)
IN2(7:0)
IN3(7:0)
IN4(7:0)
IN5(7:0)
IN6(7:0)
IN7(7:0)
HADDR(5:0)
nGGS
GGS
En
BUF
BUFFER=DEFAULT
NAME=STDBUF


Raddr(26:0)
lineaddr(6:0)
prediction(26:0)
store
predict
REGFILE
BUFFER=DEFAULT
GROUP="Predicter"
M=6
N=27
WORDS=64
NAME=REGFILE1R
BUF INV
NAME=BUFINV
BUFFER
NAME=BUFF
BUF
BUFFER=DEFAULT
NAME=STOBUF
NAME=STOBUFINV
AND2
NAME=STOAND2
NAME ADDCLA

# Line Replacing Module



# APPENDIX B. VERILOG-IN FILES

This appendix contains the verilog hardware description code for all the implemented modules of the Predicted Read Cache IC.

```
/* Finite State Machine for the snoop module, functional description in verilog
** [snoop.v] file
*/


module snoop (clk,Raddr,prty,next,back,TS,TT,CI,TBST,ARTRY,hit,en,
              AACK,Abort,flush,match,nomatch,init,addr,prtyout,A28,A27);

// epoch set_attribute FIXEDBLOCK = 1

`define encode
`define low   1'b0
`define high  1'b1

parameter // epoch enum stat
          snoop     = 4'b0000,
          ptcial    = 4'b0010,
          type      = 4'b0011,
          read      = 4'b0111,
          Yhit      = 4'b0110,
          wait1     = 4'b0100,
          nohit     = 4'b0101,
          write     = 4'b0001,
          dc_state  = 4'bxxxx;


input     [31:0] Raddr;
input     [3:0] prty;
input     [4:0] TT;
input     clk,hit,init,TS,CI,TBST,ARTRY,next,back,en;
output    AACK,Abort,flush,match,nomatch,A28,A27;
output    [27:0] addr;
output    [3:0] prtyout;

wire      [3:0] pgen;
wire      [27:0] temp;
reg       AACKEN,Abort,flush,CAR,match,nomatch;
reg       [3:0] /* epoch enum stat */ state,next_state;

supply0 GND;
supply1 VDD;
```

```
/* Data Path */

paritycgo #(8,1,"DPATH")
            PC1(Raddr[7:0],prty[0],E1,pgen[0]);
paritycgo #(8,1,"DPATH")
            PC2(Raddr[15:8],prty[1],E2,pgen[1]);
paritycgo #(8,1,"DPATH")
            PC3(Raddr[23:16],prty[2],E3,pgen[2]);
paritycgo #(8,1,"DPATH")
            PC4(Raddr[31:24],prty[3],E4,pgen[3]);
stdor4  OR1(E1,E2,E3,E4,PE);

tribuf   #(4,1,"DPATH")
     BUF1(en,pgen,prtyout);

dff_c #(28,1,"DPATH")
            CAR1(CAR,init,temp,addr);
stdtribuf BUF2(AACKEN,GND,AACK);

assign temp[26:0] = Raddr[26:0];
assign temp[27] = GND;
assign A27 = Raddr[27];
assign A28 = Raddr[28];


/* Finite State Machine */

always @(posedge clk or negedge init)
            begin
               if (!init) state= snoop;
               else state= next_state;
            end

always @(state or PE or TS or TT or CI or TBST or ARTRY or hit or back or next or
          Raddr[31:29])
            begin

              nomatch= `low;
              match  = `low;
              CAR    = `low;
              Abort  = `low;
              flush  = `low;
              AACKEN = `high;
```

```
case (state)

snoop:  begin
            if (TS==0)next_state = ptcial;
            else next_state = snoop;
        end



ptcial:  begin
            CAR = `high;
            if (PE==1)next_state = snoop;
            else next_state = type;
         end

type:    begin
            if (TBST ==0&Raddr[31:29]==3'b000&CI==1&TT==5'h1e)
                next_state = read;
            else next_state = write;
         end

write:   begin
            next_state = snoop;
            if (hit==1)flush = `high;
            else flush = `low;
         end


read:    begin
            if (hit==1) next_state = Yhit;
            else next_state = nohit;
         end

Yhit:    begin
            match = `high;
            AACKEN = `low;
            next_state= wait1;
         end

nohit:   begin
            nomatch= `high;
            next_state= wait1;
         end
```

```
wait1:      begin
                 if (ARTRY==0) Abort=`high;
                 else Abort = `low;
                 if (next | back) next_state = snoop;
                 else next_state = wait1;
             end


  default:  begin
                 next_state  = dc_state;
            end

  endcase

end

endmodule
```

```
/* Predicted Addresses Storage module, structural description in verilog
** [pbank.v] file
*/

`define numbits 28
`define group "SBank"
`define group2 "Buffers"
`define celltype 0


module pbank(PADDR,RADDR,storeAddr,init,hitbus);

// epoch set_attribute FIXEDBLOCK = 1

input     [numbits-1:0] PADDR,RADDR;
input     [15:0] storeAddr;
input     init;
output    [15:0] hitbus;

wire      [`numbits-1:0] PADDRBuf,RADDRBuf;
wire      [15:0] storeAddrBuf,local,loadU,loadL;
wire      [`numbits-1:0] SA0,SA1,SA2,SA3,SA4,SA5,SA6,
           SA7,SA8,SA9,SA10,SA11,SA12,SA13,SA14,SA15;

supply1 high;

/* Buffer inputs */

buff #(`numbits,0,`group2)
          BUF1(PADDR,PADDRBuf);
buff #(`numbits,0,`group2)
          BUF2(RADDR,RADDRBuf);
buff #(16,0,`group2)
          BUF3(storeAddr,storeAddrBuf);
buff #(16,0,`group2)
          BUF4(storeAddrBuf,loadU);
buff #(16,0,`group2)
          BUF5(storeAddrBuf,loadL);
```

```
stdbuf  BUF6(init,Lower);
stdbuf  BUF7(init,Upper);
stdbuf  BUF8(Lower,CLR1);
stdbuf  BUF9(Lower,CLR2);
stdbuf  BUF10(Lower,CLR3);
stdbuf  BUF11(Lower,CLR4);
stdbuf  BUF12(Upper,PRE1);
stdbuf  BUF13(Upper,PRE2);
stdbuf  BUF14(Upper,PRE3);
stdbuf  BUF15(Upper,PRE4);


/* This is line 0*/
dff_p #(`numbits-14,`celltype,`group)
          store0U(loadU[0],PADDRBuf[27:14],PRE1,SA0[27:14]);
dff_c #(`numbits-14,`celltype,`group)
          store0L(loadL[0],CLR1,PADDRBuf[13:0],SA0[13:0]);
equal #(`numbits,`celltype,`group)
          compare0(RADDRBuf,SA0,local[0]);

/* This is line 1*/
dff_p #(`numbits-14,`celltype,`group)
          store1U(loadU[1],PADDRBuf[27:14],PRE1,SA1[27:14]);
dff_c #(`numbits-14,`celltype,`group)
          store1L(loadL[1],CLR1,PADDRBuf[13:0],SA1[13:0]);
equal #(`numbits,`celltype,`group)
          compare1(RADDRBuf,SA1,local[1]);

/* This is line 2*/
dff_p #(`numbits-14,`celltype,`group)
          store2U(loadU[2],PADDRBuf[27:14],PRE1,SA2[27:14]);
dff_c #(`numbits-14,`celltype,`group)
          store2L(loadL[2],CLR1,PADDRBuf[13:0],SA2[13:0]);
equal #(`numbits,`celltype,`group)
          compare2(RADDRBuf,SA2,local[2]);

/* This is line 3*/
dff_p #(`numbits-14,`celltype,`group)
          store3U(loadU[3],PADDRBuf[27:14],PRE1,SA3[27:14]);
dff_c #(`numbits-14,`celltype,`group)
          store3L(loadL[3],CLR1,PADDRBuf[13:0],SA3[13:0]);
equal #(`numbits,`celltype,`group)
          compare3(RADDRBuf,SA3,local[3]);
```

```
/* This is line 4*/
dff_p #(`numbits-14,`celltype,`group)
        store4U(loadU[4],PADDRBuf[27:14],PRE2,SA4[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store4L(loadL[4],CLR2,PADDRBuf[13:0],SA4[13:0]);
equal #(`numbits,`celltype,`group)
        compare4(RADDRBuf,SA4,local[4]);

/* This is line 5*/
dff_p #(`numbits-14,`celltype,`group)
        store5U(loadU[5],PADDRBuf[27:14],PRE2,SA5[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store5L(loadL[5],CLR2,PADDRBuf[13:0],SA5[13:0]);
equal #(`numbits,`celltype,`group)
        compare5(RADDRBuf,SA5,local[5]);

/* This is line 6*/
dff_p #(`numbits-14,`celltype,`group)
        store6U(loadU[6],PADDRBuf[27:14],PRE2,SA6[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store6L(loadL[6],CLR2,PADDRBuf[13:0],SA6[13:0]);
equal #(`numbits,`celltype,`group)
        compare6(RADDRBuf,SA6,local[6]);

/* This is line 7*/
dff_p #(`numbits-14,`celltype,`group)
        store7U(loadU[7],PADDRBuf[27:14],PRE2,SA7[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store7L(loadL[7],CLR2,PADDRBuf[13:0],SA7[13:0]);
equal #(`numbits,`celltype,`group)
        compare7(RADDRBuf,SA7,local[7]);

/* This is line 8*/
dff_p #(`numbits-14,`celltype,`group)
        store8U(loadU[8],PADDRBuf[27:14],PRE3,SA8[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store8L(loadL[8],CLR3,PADDRBuf[13:0],SA8[13:0]);
equal #(`numbits,`celltype,`group)
        compare8(RADDRBuf,SA8,local[8]);
```

```
/* This is line 9*/
dff_p #(`numbits-14,`celltype,`group)
        store9U(loadU[9],PADDRBuf[27:14],PRE3,SA9[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store9L(loadL[9],CLR3,PADDRBuf[13:0],SA9[13:0]);
equal #(`numbits,`celltype,`group)
        compare9(RADDRBuf,SA9,local[9]);

/* This is line 10*/
dff_p #(`numbits-14,`celltype,`group)
        store10U(loadU[10],PADDRBuf[27:14],PRE3,SA10[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store10L(loadL[10],CLR3,PADDRBuf[13:0],SA10[13:0]);
equal #(`numbits,`celltype,`group)
        compare10(RADDRBuf,SA10,local[10]);

/* This is line 11*/
dff_p #(`numbits-14,`celltype,`group)
        store11U(loadU[11],PADDRBuf[27:14],PRE3,SA11[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store11L(loadL[11],CLR3,PADDRBuf[13:0],SA11[13:0]);
equal #(`numbits,`celltype,`group)
        compare11(RADDRBuf,SA11,local[11]);

/* This is line 12*/
dff_p #(`numbits-14,`celltype,`group)
        store12U(loadU[12],PADDRBuf[27:14],PRE4,SA12[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store12L(loadL[12],CLR4,PADDRBuf[13:0],SA12[13:0]);
equal #(`numbits,`celltype,`group)
        compare12(RADDRBuf,SA12,local[12]);

/* This is line 13*/
dff_p #(`numbits-14,`celltype,`group)
        store13U(loadU[13],PADDRBuf[27:14],PRE4,SA13[27:14]);
dff_c #(`numbits-14,`celltype,`group)
        store13L(loadL[13],CLR4,PADDRBuf[13:0],SA13[13:0]);
equal #(`numbits,`celltype,`group)
        compare13(RADDRBuf,SA13,local[13]);
```

```
/* This is line 14*/
dff_p #(`numbits-14,`celltype,`group)
          store14U(loadU[14],PADDRBuf[27:14],PRE4,SA14[27:14]);
dff_c #(`numbits-14,`celltype,`group)
          store14L(loadL[14],CLR4,PADDRBuf[13:0],SA14[13:0]);
equal #(`numbits,`celltype,`group)
          compare14(RADDRBuf,SA14,local[14]);

/* This is line 15*/
dff_p #(`numbits-14,`celltype,`group)
          store15U(loadU[15],PADDRBuf[27:14],PRE4,SA15[27:14]);
dff_c #(`numbits-14,`celltype,`group)
          store15L(loadL[15],CLR4,PADDRBuf[13:0],SA15[13:0]);
equal #(`numbits,`celltype,`group)
          compare15(RADDRBuf,SA15,local[15]);


/* Buffer Output */

buff #(16,0,`group2)
          BUF16(local,hitbus);


endmodule
```

```
/* Hit detector module, structural description in verilog
** [hitmod.v] file
*/

`define numbits 28
`define group "Bufbank"


module hitmod(PADDR,RADDR,storeAddr,init,hitbus);

// epoch set_attribute FIXEDBLOCK = 1

input    [`numbits-1:0] PADDR,RADDR;
input    [7:0] storeAddr;
input    init;
output   [127:0] hitbus;

wire     [127:0] load,LOCAL,decout;
wire     [7:0] storeAddrbuf;
wire     [`numbits-1:0] PADDRbuf1,PADDRbuf2,RADDRbuf1,RADDRbuf2;


buff #(`numbits,0,`group)
         BUF11(PADDR,PADDRbuf1);
buff #(`numbits,0,`group)
         BUF12(PADDR,PADDRbuf2);
buff #(`numbits,0,`group)
         BUF13(RADDR,RADDRbuf1);
buff #(`numbits,0,`group)
         BUF14(RADDR,RADDRbuf2);
buff #(8,0,`group)
         BUF15(storeAddr,storeAddrbuf);

stdbuf  BUF16(init,initbuf);
stdbuf  BUF17(initbuf,initbuf1);
stdbuf  BUF18(initbuf,initbuf2);
stdbuf  BUF19(initbuf,initbuf3);
stdbuf  BUF20(initbuf,initbuf4);
```

```
/* Decode load address */
decoder #(8,128,"decoder8x128.codefile")
        DEC1(storeAddrbuf,decout);

buff #(128,0,"AUTO")
        BUF21(decout,load);


/* Storage Banks */

// epoch pre_compiled pbank
pbank  BANK0(PADDRbuf1,RADDRbuf1,load[15:0],initbuf1,LOCAL[15:0]);

// epoch pre_compiled pbank
pbank  BANK1(PADDRbuf1,RADDRbuf1,load[31:16],initbuf1,LOCAL[31:16]);

// epoch pre_compiled pbank
pbank  BANK2(PADDRbuf1,RADDRbuf1,load[47:32],initbuf2,LOCAL[47:32]);

// epoch pre_compiled pbank
pbank  BANK3(PADDRbuf1,RADDRbuf1,load[63:48],initbuf2,LOCAL[63:48]);

// epoch pre_compiled pbank
pbank  BANK4(PADDRbuf2,RADDRbuf2,load[79:64],initbuf3,LOCAL[79:64]);

// epoch pre_compiled pbank
pbank  BANK5(PADDRbuf2,RADDRbuf2,load[95:80],initbuf3,LOCAL[95:80]);

// epoch pre_compiled pbank
pbank  BANK6(PADDRbuf2,RADDRbuf2,load[111:96],initbuf4,LOCAL[111:96]);

// epoch pre_compiled pbank
pbank  BANK7(PADDRbuf2,RADDRbuf2,load[127:112],initbuf4,LOCAL[127:112]);


/* Buffer Output */

buff #(128,0,`group)
        BUF22(LOCAL,hitbus);

endmodule
```

```
/* 8 x 3 encoder module, structural description in verilog
** [encoder.v]
*/

module encoder8x3(IN,EN,LADDR,GSL,GSH);

// epoch set_attribute FIXEDBLOCK = 0

input     [7:0] IN;
input     EN;
output    [2:0] LADDR;
output    GSL,GSH;

wire      [7:0] BIN;
supply1   VDD;

/* find out if "got something" */
stdor4 U1 (IN[0],IN[1],IN[2],IN[3],GSL);
stdor4 U2 (IN[4],IN[5],IN[6],IN[7],GSH);

/* Buffer and invert inputs*/
buff #(7,0,"AUTO") U3 (IN[7:1],BIN[7:1]);
stdinv U4 (BIN[6],bin6);
stdinv U5 (BIN[5],bin5);
stdinv U6 (BIN[4],bin4);
stdinv U7 (BIN[2],bin2);

/* Encode with bit7 highest priority */

stdnor4 U8 (BIN[4],BIN[5],BIN[6],BIN[7],addrlow2);
        stdinvtri U9 (EN,addrlow2,LADDR[2]);

stdaoi3333 U10 (BIN[2],bin4,bin5,BIN[3],bin4,bin5,
       BIN[6],VDD,VDD,BIN[7],VDD,VDD,addrlow1);
        stdinvtri U11 (EN,addrlow1,LADDR[1]);

stdaoi4444 U12 (BIN[1],bin2,bin4,bin6,BIN[3],bin4,bin6,VDD,
       BIN[5],bin6,VDD,VDD,BIN[7],VDD,VDD,VDD,addrlow0);
        stdinvtri U13 (EN,addrlow0,LADDR[0]);

endmodule
```

```
/* 64 x 6 encoder module, structural description in verilog
** This makes use of 8 instances of encoder8x3 module
** [encoder.v]
*/

module encoder64x6 (IN,GEN,HADDR,GGS,nggs);


// epoch set_attribute FIXEDBLOCK = 0

input       [63:0] IN;
input       GEN;
output      [5:0] HADDR;
output      GGS,nggs;

supply1 VDD;

/* find out if group "got something" */

encoder8x3  enc0(IN[7:0],EN0,HADDR[2:0],GSL0,GSH0);
encoder8x3  enc1(IN[15:8],EN1,HADDR[2:0],GSL1,GSH1);
encoder8x3  enc2(IN[23:16],EN2,HADDR[2:0],GSL2,GSH2);
encoder8x3  enc3(IN[31:24],EN3,HADDR[2:0],GSL3,GSH3);
encoder8x3  enc4(IN[39:32],EN4,HADDR[2:0],GSL4,GSH4);
encoder8x3  enc5(IN[47:40],EN5,HADDR[2:0],GSL5,GSH5);
encoder8x3  enc6(IN[55:48],EN6,HADDR[2:0],GSL6,GSH6);
encoder8x3  enc7(IN[63:56],EN7,HADDR[2:0],GSL7,GSH7);

stdor4 U14 (GSL0,GSL1,GSL2,GSL3,gshl1);
stdor4 U15 (GSL4,GSL5,GSL6,GSL7,gshl2);
stdor4 U16 (GSH0,GSH1,GSH2,GSH3,gshl3);
stdor4 U17 (GSH4,GSH5,GSH6,GSH7,gshl4);
stdor4 U18 (gshl1,gshl2,gshl3,gshl4,GGS);
stdnor4 U19 (gshl1,gshl2,gshl3,gshl4,nggs);

/* If group is enabled, enable appropriate subgroup
   subgroup enc7 has higher priority. */

// enable enc7 if it got something
stdor2  U20 (GSL7,GSH7,gs7);
stdnor2 U21 (GSL7,GSH7,ngs7);
stdand2 U22 (gs7,GEN,EN7);
```

```
// enable enc6 if not.gs7,but gs6
stdor2  U23 (GSL6,GSH6,gs6);
stdnor2 U24 (GSL6,GSH6,ngs6);
stdand2 U25 (ngs7,gs6,tmp6);
stdand2 U26 (tmp6,GEN,EN6);

// enable enc5 if not.gs7 and not.gs6, but gs5
stdor2  U27 (GSL5,GSH5,gs5);
stdnor2 U28 (GSL5,GSH5,ngs5);
stdand3 U29 (ngs7,ngs6,gs5,tmp5);
stdand2 U30 (tmp5,GEN,EN5);

// enable enc4 (by same reasoning as above)
stdor2  U31 (GSL4,GSH4,gs4);
stdnor2 U32 (GSL4,GSH4,ngs4);
stdand3 U33 (ngs7,ngs6,gs4,tmp4);
stdand3 U34 (tmp4,ngs5,GEN,EN4);

// enable enc3
stdor2  U35 (GSL3,GSH3,gs3);
stdnor2 U36 (GSL3,GSH3,ngs3);
stdand4 U37(ngs7,ngs6,ngs5,gs3,tmp3);
stdand3 U38 (tmp3,ngs4,GEN,EN3);

// enable enc2
stdor2  U39 (GSL2,GSH2,gs2);
stdnor2 U40 (GSL2,GSH2,ngs2);
stdand4 U41 (ngs7,ngs6,ngs5,ngs4,tmp);
stdand2 U42 (ngs3,gs2,tmp2);
stdand3 U43 (tmp,tmp2,GEN,EN2);

// enable enc1
stdor2  U44 (GSL1,GSH1,gs1);
stdnor2 U45 (GSL1,GSH1,ngs1);
stdand3 U46 (ngs3,ngs2,gs1,tmp1);
stdand3 U47 (tmp,tmp1,GEN,EN1);


// enable enc0
stdor2  U48 (GSL0,GSH0,gs0);
stdand4 U49 (ngs3,ngs2,ngs1,gs0,tmp0);
stdand3 U50 (tmp,tmp0,GEN,EN0);
```

```
/* Encode address of selected subgroup */
// enc7 highest priority
 stdnor4 U51 (gs4,gs5,gs6,gs7,addrhigh2);
            stdinvtri U52 (GEN,addrhigh2,HADDR[5]);

stdaoi3333 U53 (gs2,ngs4,ngs5,gs3,ngs4,ngs5,
          gs6,VDD,VDD,gs7,VDD,VDD,addrhigh1);
            stdinvtri U54 (GEN,addrhigh1,HADDR[4]);

stdaoi4444 U55 (gs1,ngs2,ngs4,ngs6,gs3,ngs4,ngs6,VDD,
           gs5,ngs6,VDD,VDD,gs7,VDD,VDD,VDD,addrhigh0);
            stdinvtri U56 (GEN,addrhigh0,HADDR[3]);

endmodule

/* 128 x 7 encoder module, structural description in verilog
** This makes use of 2 instances of encoder64x6 module
** [encmod.v]
*/
module encoder (IN,HITADDR,HIT);

// epoch set_attribute FIXEDBLOCK = 1

input     [127:0] IN;
output    [6:0] HITADDR;
output    HIT;

/* find out if there is a hit */
encoder64x6  group0(IN[63:0],GEN0,HITADDR[5:0],GGS0,nggs0);
encoder64x6  group1(IN[127:64],GEN1,HITADDR[5:0],GGS1,nggs1);
stdor2 U57 (GGS0,GGS1,HIT);

/* Enable group, group1 has higher priority */
// enable group1 if it got something
stdbuf U58 (GGS1,GEN1);

// enable group2 if not.GGS1, but GGS0
stdand2 U59 (nggs1,GGS0,GEN0);

/* Find Hit Address msb*/
assign HITADDR[6] = GGS1;

endmodule
```

```
/* Predict module, structural description in verilog
** [Predict.v] file
*/

`define group "Predicter"
`define group2 "Buffers"


module predict(Raddr,lineaddr,store,predict,prediction);

// epoch set_attribute FIXEDBLOCK = 1

input       [26:0] Raddr;
input       [6:0] lineaddr;
input       store,predict;
output      [26:0]prediction;

wire        [27:0] operandA,operandB,Result;
wire        [26:0] invaddr;
wire        [5:0] line,Rline,Wline;

supply0 grnd;
supply1 VDD;



/* Buffer and invert some inputs */
bufinv #(27,0,`group2)
            addr_buf (Raddr,invaddr);
buff #(6,0,`group2)
            line_buff (lineaddr[5:0],line);
buff #(6,0,`group2)
            Read_buff (line,Rline);
buff #(6,0,`group2)
            write_buff (line,Wline);

stdbuf    buf1 (lineaddr[6],highreg);

stdbufinv  INV1 (lineaddr[6],lowreg);
```

```
/* Create a 128-word regfile, using two regfiles of 64 words each */

regfile1r #(27,64,6,`group)
          register0(invaddr,Rline,RE0,Wline,WE0,operandB[26:0]);
regfile1r #(27,64,6,`group)
          register1(invaddr,Rline,RE1,Wline,WE1,operandB[26:0]);


assign operandB[27]= grnd;

/* Decode enable signals */
stdand2  ANDr1(highreg,predict,E1);
stdbuf   buf2 (E1,RE1);
stdand2  ANDr0(lowreg,predict,E2);
stdbuf   buf3 (E2,RE0);
stdand2  ANDw1(highreg,store,E3);
stdbuf   buf4 (E3,WE1);
stdand2  ANDw0(lowreg,store,E4);
stdbuf   buf5 (E4,WE0);

/* multiply Operand A by 2 */
assign operandA[27:1] = Raddr;
assign operandA[0] = grnd;

/* Predict value */
addcla #(28,0,`group)
          predicter (operandA,operandB,VDD,cout,Result);
assign prediction = Result[26:0];

endmodule
```

```
/* Finite State Machine for predicter module, functional description in verilog
** [pmfsm.v] file
*/

module pmfsm (clk,status,match,nomatch,init,next,back,par,add,pred);

// epoch set_attribute FIXEDBLOCK = 1

`define encode
`define low   1'b0
`define high  1'b1

parameter // epoch enum stat
            idle0      = 5'b00000,
            stnby1     = 5'b00010,
            sum1       = 5'b00011,
            wait1      = 5'b00111,
            store2     = 5'b00110,
            stnby2     = 5'b00100,
            store1     = 5'b00101,
            Back1      = 5'b00001,
            store3     = 5'b01001,
            Next       = 5'b01000,
            store4     = 5'b01100,
            wait2      = 5'b01101,
            sum2       = 5'b01111,
            stnby3     = 5'b01011,
            idle1      = 5'b01010,
            Back2      = 5'b01110,
            dc_state   = 5'bxxxxx;

input       [1:0] status;
input       clk,match,nomatch,init;
output      next,back,par,add,pred;


reg         next,back,flag,next_flag,par,pred,add;
reg         [4:0] /* epoch enum stat */ state,next_state;
```

```
always @(posedge clk or negedge init)
        begin
          if (!init)
                begin
                  flag = `low;
                  state= idle0;
                end
          else
                begin
                  flag = next_flag;
                  state= next_state;
                end
          end
always @(state or status or match or nomatch or flag )
        begin
          next = `low;
          par  = `low;
          pred = `low;
          back = `low;
          add  = `low;
          next_flag = flag;

          case (state)

          idle0: begin
                  if (match != nomatch)
                      begin
                        if (match) next_state = stnby1;
                        else next_state = stnby2;
                      end
                  else
                        next_state = idle0;
                end

          stnby1:  begin
                      if (status[1] == 0) next_state = sum1;
                      else  next_state = stnby1;
                   end

          sum1:   begin
                      add = `high;
                      next_state = wait1;
                  end
```

```
wait1:  begin
            add = `high;
            next_state = store1;
        end

store1: begin
            add = `high;
            par = `high;
            pred= `high;
            next_state = Back1;
        end

Back1:  begin
            back = `high;
            next_state = idle0;
        end

stnby2: begin
            if (status[1] == 0) next_state = store2;
            else  next_state = stnby2;
        end

store2: begin
            par = `high;
            next_state = Back2;
        end

Back2:  begin
            back = `high;
            next_state = idle1;
        end

idle1: begin
        if (match != nomatch)
            begin
              next_state = stnby3;
              if (match) next_flag = `high;
              else next_flag = `low;
            end
        else
            next_state = idle1;
        end
```

```
                stnby3: begin
                              if (status[1] == 0) next_state = sum2;
                              else  next_state = stnby3;
                         end

                sum2:  begin
                              add = `high;
                              next_state = wait2;
                         end

                wait2:  begin
                              add = `high;
                              if (flag == 1) next_state = store4;
                              else  next_state = store3;
                         end

                store4:  begin
                              add = `high;
                              par = `high;
                              pred= `high;
                              next_state = Back2;
                         end

                store3:  begin
                              add = `high;
                              par = `high;
                              pred= `high;
                              next_state = Next;
                         end

                Next:    begin
                              next = `high;
                              next_state = idle0;
                         end

                default:  begin
                              next_state  = dc_state;
                              next_flag = 1'bx;
                         end
                endcase
           end
endmodule
```

```
/* Replacing Algorithm module, structural description in verilog
** [linerep.v] file
*/

module linerep (clk,Lnum,match,next,back,init,status,line);

// epoch set_attribute FIXEDBLOCK = 1

`define encode
`define low 1'b0
`define high  1'b1
`define valid_ready      2'b00
`define valid_notready   2'b01
`define notvalid_ready   2'b10
`define notvalid_notready 2'b11

parameter // epoch enum stat
            ready       = 4'b0000,
            load1       = 4'b0010,
            Count       = 4'b0011,
            read        = 4'b0001,
            inspt       = 4'b0101,
            clear       = 4'b0111,
            tmp1        = 4'b0100,
            wait1       = 4'b1000,
            load2       = 4'b1010,
            set         = 4'b1011,
            tmp2        = 4'b1111,
            tmp3        = 4'b1110,
            Back        = 4'b1100,
            dc_state= 4'bx;

input       [6:0] Lnum;
input       clk,match,back,next,init;
output      [1:0] status;
output      [6:0] line;

reg         flagw,flagr,count,new,Lhit;
reg         sel1,sel2,next_sel1,next_sel2;
reg         [1:0] status, next_status;
reg         [3:0] /*epoch enum stat */ state,next_state;
```

```
wire        [6:0] muxout,oldcount,newcount,ROMline,HITline;

supply0 GND;
supply1 VDD;


/* Data Path */

stdbuf buf1(init,initbuf);

rom #(7,128,7,"counter.codefile",8)
            counter (oldcount,newcount);
dff_c #(7,1,"DATAP")
            feedreg (count,initbuf,newcount,oldcount);
dff_c #(7,1,"DATAP")
            ROMreg(new,initbuf,newcount,ROMline);
dff_c #(7,1,"DATAP")
            HITreg(Lhit,initbuf,Lnum,HITline);
mux2  #(7,1,"DATAP")
            Lmux (ROMline,HITline,sel2,line);
equal #(7,1,"DATAP")
            compare(ROMline,HITline,comp);
mux2  #(7,1,"DATAP")
            Fmux (newcount,Lnum,sel1,muxout);
stdmux2 Bmux (GND,VDD,sel1,bit);
hsramoe #(1,128,7,2,0)
            FLAGS(muxout,bit,flagr,flagw,flag);


/* State Machine */

always @(posedge clk or negedge initbuf)
 begin
   if (!initbuf)
    begin
             state  =  ready;
      status = `valid_ready;
             sel1   = `low;
             sel2   = `low;
    end
   else
    begin
      state  = next_state;
```

```
            status = next_status;
            sel1   = next_sel1;
            sel2   = next_sel2;
       end
    end


always @(state or next or back or match or flag or comp or status or sel1 or sel2)
  begin
    count = `low;
    new   = `low;
    Lhit  = `low;
    flagr = `high;
    flagw = `high;

    next_sel1 = sel1;
    next_sel2 = sel2;
    next_status = status;

    case (state)

          ready:  begin
                     if (match != next)
                       begin
                         next_status = `notvalid_notready;
                         if (next) next_state = load1;
                         else
                           begin
                               next_state = wait1;
                               next_sel1  = `high;
                               next_sel2  = `high;
                           end
                        end
                     else
                        next_state = ready;
                  end

          load1:  begin
                        new   = `high;
                        flagw = `low;
                        next_state = Count;
                        next_status= `valid_notready;
                  end
```

```
Count:    begin
            count = `high;
            next_state = read;
          end


read:     begin
            flagr = `low;
            next_state = inspt;
          end

inspt:    begin
            flagr = `low;
            if (flag) next_state = clear;
            else next_state = tmp1;
          end

clear:    begin
            flagw = `low;
            next_state = Count;
          end

tmp1:     begin
            next_state = ready;
            next_status = `valid_ready;
            next_sel2 = `low;
          end

wait1:    next_state = load2;


load2:    begin
            Lhit = `high;
            next_status= `valid_notready;
            next_state = set;
          end

set:      begin
            flagw = `low;
            if (comp)
              begin
                 next_sel1 = `low;
                 next_state= Count;
```

```
                    end
                  else   next_state= tmp2;
                end

        tmp2:   next_state = tmp3;


        tmp3:    next_state = Back;


        Back:    begin
                  if (back)
                    begin
                        next_sel1 = `low;
                        next_state= tmp1;
                    end
                  else next_state= Back;
                 end

        default: begin
                    next_state  = dc_state;
                    next_status = 2'bx;
                    next_sel1 = 1'bx;
                    next_sel2 = 1'bx;
                 end

    endcase

  end

endmodule
```

```
/* ROM.codefile, Functional description in verilog
** [counter.codefile] file
*/
// PLA TABLE
// output
0000001   // position 0000000
0000010   // position 0000001
0000011
0000100
0000101
0000110
0000111
0001000
0001001
0001010
0001011
0001100
0001101
0001110
0001111
0010000
0010001
0010010
0010011
0010100
0010101
0010110
0010111
0011000
0011001
0011010
0011011
0011100
0011101
0011110
0011111
0100000
0100001
0100010
0100011
0100100
```

0100101
0100110
0100111
0101000
0101001
0101010
0101011
0101100
0101101
0101110
0101111
0110000
0110001
0110010
0110011
0110100
0110101
0110110
0110111
0111000
0111001
0111010
0111011
0111100
0111101
0111110
0111111
1000000
1000001
1000010
1000011
1000100
1000101
1000110
1000111
1001000
1001001
1001010
1001011
1001100
1001101
1001110
1001111

1010000
1010001
1010010
1010011
1010100
1010101
1010110
1010111
1011000
1011001
1011010
1011011
1011100
1011101
1011110
1011111
1100000
1100001
1100010
1100011
1100100
1100101
1100110
1100111
1101000
1101001
1101010
1101011
1101100
1101101
1101110
1101111
1110000
1110001
1110010
1110011
1110100
1110101
1110110
1110111
1111000
1111001
1111010

```
1111011
1111100
1111101
1111110
1111111    // position 1111110
0000000    // position 1111111


// END TABLE
```

```
/* Decoder8x128.codefile, Functional description in verilog
** [Decoder8x128.codefile] file
*/
// PLA TABLE
//  sel0 sel1 sel2 sel3 sel4 sel5 sel6 EN
 00000001              // line 0
 10000001              // line 1
 01000001              // line 2
 11000001              // line 3
 00100001              // line 4
 10100001              // line 5
 01100001              // line 6
 11100001              // line 7
 00010001              // line 8
 10010001              // line 9
 01010001              // line 10
 11010001              // line 11
 00110001              // line 12
 10110001              // line 13
 01110001              // line 14
 11110001              // line 15
 00001001              // line 16
 10001001              // line 17
 01001001              // line 18
 11001001              // line 19
 00101001              // line 20
 10101001              // line 21
 01101001              // line 22
 11101001              // line 23
 00011001              // line 24
 10011001              // line 25
 01011001              // line 26
 11011001              // line 27
 00111001              // line 28
 10111001              // line 29
 01111001              // line 30
 11111001              // line 31
 00000101              // line 32
 10000101              // line 33
 01000101              // line 34
 11000101              // line 35
```

```
00100101        // line 36
10100101        // line 37
01100101        // line 38
11100101        // line 39
00010101        // line 40
10010101        // line 41
01010101        // line 42
11010101        // line 43
00110101        // line 44
10110101        // line 45
01110101        // line 46
11110101        // line 47
00001101        // line 48
10001101        // line 49
01001101        // line 50
11001101        // line 51
00101101        // line 52
10101101        // line 53
01101101        // line 54
11101101        // line 55
00011101        // line 56
10011101        // line 57
01011101        // line 58
11011101        // line 59
00111101        // line 60
10111101        // line 61
01111101        // line 62
11111101        // line 63
00000011        // line 64
10000011        // line 65
01000011        // line 66
11000011        // line 67
00100011        // line 68
10100011        // line 69
01100011        // line 70
11100011        // line 71
00010011        // line 72
10010011        // line 73
01010011        // line 74
11010011        // line 75
00110011        // line 76
10110011        // line 77
01110011        // line 78
```

```
11110011            // line 79
00001011            // line 80
10001011            // line 81
01001011            // line 82
11001011            // line 83
00101011            // line 84
10101011            // line 85
01101011            // line 86
11101011            // line 87
00011011            // line 88
10011011            // line 89
01011011            // line 90
11011011            // line 91
00111011            // line 92
10111011            // line 93
01111011            // line 94
11111011            // line 95
00000111            // line 96
10000111            // line 97
01000111            // line 98
11000111            // line 99
00100111            // line 100
10100111            // line 101
01100111            // line 102
11100111            // line 103
00010111            // line 104
10010111            // line 105
01010111            // line 106
11010111            // line 107
00110111            // line 108
10110111            // line 109
01110111            // line 110
11110111            // line 111
00001111            // line 112
10001111            // line 113
01001111            // line 114
11001111            // line 115
00101111            // line 116
10101111            // line 117
01101111            // line 118
11101111            // line 119
00011111            // line 120
10011111            // line 121
```

```
01011111        // line 122
11011111        // line 123
00111111        // line 124
10111111        // line 125
01111111        // line 126
11111111        // line 127


// END TABLE
```

# APPENDIX C. VERILOG TESTSHELL FILES

This appendix contains the verilog testeshell code to simulate all the implemented modules of the Predicted Read Cache IC.

```
//--------------------------------------------
// File         : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2/vout/snoop.i.v
// Date          : Fri Feb 10 15:39:24 1995
// Program        : vrlgout 1.1
// Project       : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2
// Design        : snoop
//--------------------------------------------

`resetall
`timescale 1ns / 1ps

module snoop_testshell;

   reg  clk,TS,CI,TBST,ARTRY,hit,en,next,back,init;
   reg  [31:0] Raddr;
   reg  [3:0] prty;
   reg  [4:0] TT;
   wire PE,AACK,Abort,flush,CAR,match,nomatch;
   wire [2:0] snum;
   wire [3:0] prtyout;
   wire [27:0] addr;

  snoop  inst_snoop (.clk(clk), .match(match), .nomatch(nomatch), .init(init), .TS(TS),
    .CI(CI), .TBST(TBST), .ARTRY(ARTRY), .hit(hit), .en(en), .next(next),
    .back(back), .PE(PE), .AACK(AACK), .TT(TT), .Abort(Abort),
    .flush(flush), .CAR(CAR), .Raddr(Raddr), .addr(addr), .prty(prty),
    .prtyout(prtyout), .snum(snum));

initial
  begin
// start tasks
  Waves;
  Monitor;

// test data

  clk = 0;
  init= 1;

  #2
  init= 0;
```

```
#4
init= 1;

#4
Raddr= 32'h00000001;
prty= 4'b1111;
TS = 0;
TT = 5'h0e;
CI = 1;

#15
TS = 1;

#2
TS = 1;

#50
Raddr= 32'h00000002;
prty= 4'b1110;
TS = 0;
TT = 5'h0e;
CI = 1;

#15
TS = 1;

#2
TS = 1;


#50
Raddr= 32'h00000003;
prty= 4'b1111;
TS = 0;
TT = 5'h0e;
CI = 1;

#15
TS = 1;

#15
hit = 0;
```

```
#15
hit = 1;

#15
hit = 0;


#50
Raddr= 32'h00000004;
prty= 4'b1110;
TS = 0;
TT = 5'h1e;
CI = 1;
TBST= 0;

#15
TS = 1;
back = 0;
next = 0;

#2
TS = 1;

#30
back = 1;

#50
Raddr= 32'h00000005;
prty= 4'b1111;
TS = 0;
TT = 5'h1e;
CI = 1;
TBST= 0;

#15
TS = 1;

#15
hit = 0;

#15
hit = 1;

#15
```

```
  hit = 0;
  next=0;
  ARTRY = 0;

  #20
  next = 1;
  ARTRY = 1;

  #15
  next = 0;

  #50
  Raddr= 32'h00000000;
   $display("*** DONE ***");

$stop;
$finish;

end  /* initial */

always #7.5 clk= ~clk;


// define tasks
// ---------------------------------------------------------------------

task Waves;

 $gr_waves ("clk%h",clk,"CAR%b",CAR,"match%b",match,"nomatch%b",nomatch,
            "flush%b",flush,"Abort%b",Abort,"PE%b",PE,"AACK%b",AACK,
            "addr%h",addr,"snum%h",snum);
endtask

task Monitor;
begin

  $fmonitor(1,$time," clk=%h ",clk," CAR%b ",CAR," match%b ",match,"
             nomatch%b ",nomatch," flush%b ",flush," Abort%b ",Abort,
          " PE%b ",PE," AACK%b ",AACK," addr%h ",addr," snum%h ",snum);
end
endtask

endmodule
```

```
//--------------------------------------------
// File        : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2/vout/pbank.i.v
// Date        : Fri Feb 10 15:39:24 1995
// Program     : vrlgout 1.1
// Project     : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2
// Design      : pbank
//--------------------------------------------

`resetall
`timescale 1ns / 1ps

module pbank_testshell;

   reg  [27:0] RADDR,PADDR;
   reg  [15:0] storeAddr;
   reg  init;
   wire [15:0] hitbus;


  pbank   inst_pbank (.RADDR(RADDR), .storeAddr(storeAddr), .init(init),
                      .hitbus(hitbus), .PADDR(PADDR));

initial

  begin
// start tasks
  Waves;
  Monitor;

// test data

  RADDR = 28'H0000001;

  #2
  init = 1;
  storeAddr = 16'H0000;

  #1
  init = 0;

  #5
```

```
init = 1;

#2
RADDR= 28'H0000002;

#4
PADDR= 28'H0000003;

#2
storeAddr= 16'H0001;

#2
storeAddr= 16'H0000;

#4
RADDR = 28'H0000003;

#4
PADDR= 28'H0000005;

#2
storeAddr= 16'H8000;

#4
RADDR = 28'H0000005;
storeAddr= 16'H0000;

#8
RADDR = 28'H0000007;

#4
PADDR = 28'H0000007;

#2
storeAddr= 16'H0002;

#4
RADDR = 28'H0000007;
storeAddr= 16'H0000;

#6
RADDR = 28'H0000007;
```

```
  $display("*** DONE ***");

$stop;
$finish;

end  /* initial */


// define tasks
// ----------------------------------------------------------------------

task Waves;

  $gr_waves ("RADDR%h",RADDR,"PADDR%h",PADDR,
                "init%b",init,"storeAddr%h",storeAddr,
               "hitbus%h",hitbus);
endtask


task Monitor;
begin

  $fmonitor(1,$time," RADDR%h ",RADDR," PADDR%h ",PADDR,
                " init%b ",init," storeAddr%h ",storeAddr,
               " hitbus%h ",hitbus);
end
endtask

endmodule
```

```
//--------------------------------------------
// File         : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2/vout/hitmod.i.v
// Date         : Fri Feb 10 15:39:24 1995
// Program      : vrlgout 1.1
// Project      : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2
// Design       : hitmod
//--------------------------------------------

`resetall
`timescale 1ns / 1ps

module hitmod_testshell;

   reg  [27:0] RADDR,PADDR;
   reg  [7:0] storeAddr;
   reg  init;
   wire [127:0] hitbus;


  hitmod   inst_hitmod (.RADDR(RADDR), .storeAddr(storeAddr), .init(init),
                        .hitbus(hitbus), .PADDR(PADDR));

initial

  begin
// start tasks
  Waves;
  Monitor;

// test data

  RADDR = 28'H0000001;

  #2
  init = 1;
  storeAddr = 8'H00;

  #1
  init = 0;

  #8
```

```
init = 1;

#2
RADDR= 28'H0000002;

#4
PADDR= 28'H0000003;

#2
storeAddr= 8'H80;

#4
storeAddr= 8'H00;

#6
RADDR = 28'H0000003;

#8
PADDR= 28'H0000005;

#2
storeAddr= 8'Hff;

#4
storeAddr= 8'H00;

#8
RADDR = 28'H0000005;

#8
PADDR = 28'H0000007;

#2
storeAddr= 8'H81;

#4
storeAddr= 8'H00;

#6
RADDR = 28'H0000007;

#8
PADDR = 28'H0000007;
```

```
  #2
  storeAddr= 8'Hff;

  #4
  storeAddr= 8'H00;

  #6
  RADDR = 28'H0000007;


  #6
  RADDR = 28'H0000008;


   $display("*** DONE ***");

$stop;
$finish;

end  /* initial */


// define tasks
// ---------------------------------------------------------------------

task Waves;

  $gr_waves ("RADDR%h",RADDR,"PADDR%h",PADDR,
                 "init%b",init,"storeAddr%h",storeAddr, "hitbus%h",hitbus);
endtask


task Monitor;
begin

  $fmonitor(1,$time," RADDR%h ",RADDR," PADDR%h ",PADDR,
                 " init%b ",init," storeAddr%h ",storeAddr," hitbus%h ",hitbus);
end
endtask

endmodule
```

```
//-------------------------------------------
// File        : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2/vout/encoder.i.v
// Date        : Fri Feb 10 15:39:24 1995
// Program     : vrlgout 1.1
// Project     : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2
// Design      : encoder
//-------------------------------------------

`resetall
`timescale 1ns / 1ps

module encoder_testshell;

   reg  [127:0] hitbus;
   wire [6:0] hitaddr;
   wire hit;

  encoder   inst_encoder (.IN(hitbus), .HITADDR(hitaddr), .HIT(hit));

initial

  begin
// start tasks
  Waves;
  Monitor;

// test data

   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'Hffffffffffffffff;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'H0000000000000000;
```

```
   #10
   hitbus= 128'Hffffffffffffffff;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'Hfffffffffffffffffffff;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'Hffffffffffffffffffffffffffffff;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'H0000000000000000;

   #10
   hitbus= 128'H0000000000000000;

  $display("*** DONE ***");

$stop;
$finish;

end /* initial */
```

```
// define tasks
// ----------------------------------------------------------------------

task Waves;

  $gr_waves ("hitbus%h",hitbus,"hitaddr%h",hitaddr,"hit%b",hit);
endtask


task Monitor;
begin

  $fmonitor(1,$time," hitbus=%h ",hitbus," hitaddr=%h ",hitaddr," hit=%b ",hit);

end
endtask

endmodule
```

```
//--------------------------------------------
// File         : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2/vout/predict.i.v
// Date         : Fri Feb 10 15:39:24 1995
// Program      : vrlgout 1.1
// Project      : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2
// Design       : predict
//--------------------------------------------

`resetall
`timescale 1ns / 1ps

module PDDRbank_testshell;

   reg  [27:0] RADDR,RADDR;
   reg  [6:0] storeAddr;
   reg  init;
   wire [16:0] hitbus;


  PADDRbank   inst_PADDRbank (.Raddr(Raddr), .lineaddr(lineaddr),
                    .predict(predict), .prediction(prediction), .store(store));

initial

  begin
// start tasks
  Waves;
  Monitor;

// test data

  lineaddr = 7'H00;
  store = 0;
  predict = 0;

  #4;
  Raddr = 27'H0000001;

  #2;
  store = 1;
```

```
#2;
store = 0;

#4;
Raddr = 27'H0000002;

#2;
predict = 1;

#5;
predict = 0;
store = 1;

#2;
store = 0;

#4;
Raddr = 27'H0000003;

#2;
predict = 1;

#5;
predict = 0;
store = 1;

#2;
store = 0;

#4;
Raddr = 27'H0000005;

#2;
predict = 1;


#5;
predict = 0;
store = 1;

#2;
store = 0;
```

```
#4;
Raddr = 27'H0000000;

#2;
predict = 1;


#5;
predict = 0;
store = 1;

#2;
store = 0;

#4;
Raddr = 27'H00fffff;

#2;
predict = 1;

#5;
predict = 0;
store = 1;

#2;
store = 0;

#4;
Raddr = 27'Hfffffffd;
lineaddr = 7'H01;

#2;
store = 1;


#2;
store = 0;

#4;
Raddr = 27'Hfffffff;

#2;
predict = 1;
```

```
  #5;
  predict = 0;
  store = 1;

  #2;
  store = 0;

   $display("*** DONE ***");

$stop;
$finish;

end /* initial */


// define tasks
// -----------------------------------------------------------------------

task Waves;

  $gr_waves ("RADDR%h",Raddr,"lineADDR%h",lineaddr,
                  "store%b",store,"predict%b",predict ,"prediction%h",prediction);
endtask


task Monitor;
begin

  $fmonitor(1,$time," RADDR%h ",Raddr," lineADDR%h ",lineaddr,
                " store%b ",store," predict%b ",predict ," prediction%h ",prediction);
end
endtask

endmodule
```

```
//--------------------------------------------
// File        : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2/vout/pmfsm.i.v
// Date        : Fri Feb 10 15:39:24 1995
// Program     : vrlgout 1.1
// Project     : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2
// Design      : pmfsm
//--------------------------------------------

`resetall
`timescale 1ns / 1ps

module pmfsm_testshell;

   reg  clk,match,nomatch,init;
   reg  [1:0] status;
   wire back,next,par,add,pred,flag;
   wire [3:0] snum;

  pmfsm   inst_pmfsm (.clk(clk), .match(match), .nomatch(nomatch),
                      .init(init), .next(next), .par(par), .add(add), .pred(pred),
                      .snum(snum), .flag(flag), .status(status), .back(back));

initial

  begin
// start tasks
  Waves;
  Monitor;

// test data

  clk = 0;
  init= 1;
  match = 0;
  nomatch = 0;
  status = 2'b00;

  #5
  init= 0;

  #5
```

```
init= 1;

#5
match =1;

#5
status = 2'b01;

#10
match =0;

#80
status = 2'b11;

#20
nomatch =1;

#15
nomatch =0;

#5
status = 2'b01;

#80
status = 2'b11;

#20
match =1;

#5
status = 2'b01;

#10
match =0;

#80
status = 2'b11;

#20
nomatch =1;

#5
status = 2'b01;
```

```
  #10
  nomatch =0;

  #100
  match =1;
  nomatch =1;


  #15
  match =0;
  nomatch = 0;


    $display("*** DONE ***");

$stop;
$finish;

end  /* initial */
always #7.5 clk= ~clk;

// define tasks
// ---------------------------------------------------------------------

task Waves;

  $gr_waves ("clk%h",clk,"status%b",status,"match%b",match,
    "nomatch%b",nomatch,"init%b",init,"next%b",next,"back%b",back,
"par%b",par,"add%b",add,"pred%b",pred,"flag%b",flag,"state%h",snum);
endtask

task Monitor;
begin

  $fmonitor(1,$time," clk=%h ",clk," status=%b ",status," match=%b ",match,
   " nomatch=%b ",nomatch," init=%b ",init," next=%b ",next," back%b ",back,
   " par=%b ",par," add=%b ",add," pred= %b ",pred," flag=%b ",flag,
   " state=%b ",snum);
end
endtask

endmodule
```

```
//--------------------------------------------
// File          : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2/vout/linerep.i.v
// Date         : Sat Mar 11 17:29:46 1995
// Program   : vrlgout 1.1
// Project     : /tmp_mnt/h/kepler_u0/meaguila/projects/PRC2
// Design      : linerep
//--------------------------------------------

`resetall
`timescale 1ns / 1ps

module linerep_testshell;

   reg  [6:0] Lnum;
   reg  back;
   reg  clk;
   reg  init;
   wire [6:0] line;
   reg  match;
   reg  next;
   wire [1:0] status;

  linerep   inst_linerep (.Lnum(Lnum), .back(back), .clk(clk), .init(init), .line(line),
                       .match(match), .next(next), .status(status));
initial

  begin
// start tasks
  Waves;
  Monitor;

// test data

  clk = 0;
  next= 0;
  init= 1;
  back = 0;
  match= 0;
  Lnum= 7'H00;

  #2
```

```
  init = 0;


  #15
  init = 1;

  #10
  match = 1;

  #6
  Lnum= 7'h04;

  #9
  match = 0;

  #75
  back = 1;

  #15
  back = 0;

 init = 1;
  while (line[3] == 1'b0)
     begin
           #120 next = 1;
           #15  next = 0;
     end



   $display("*** DONE ***");

$stop;
$finish;

end /* initial */

always #7.5 clk= ~clk;


// define tasks
// ------------------------------------------------------------------------
```

```
task Waves;

  $gr_waves ("clk%h",clk,"match%b",match,"next%b",next,"back%b",back,
          "init%b",init,"Lnum%h",Lnum,"status%b",status,"line%h",line);
endtask


task Monitor;
begin

  $fmonitor(1,$time," clk=%h ",clk," match%b ",match," next%b ",next,
" back%b ",back," init%b ",init," Lnum%h ",Lnum," status%b ",status," line%h ",line);
end
endtask


endmodule
```

# APPENDIX D. RPB PIN-OUT DIAGRAM

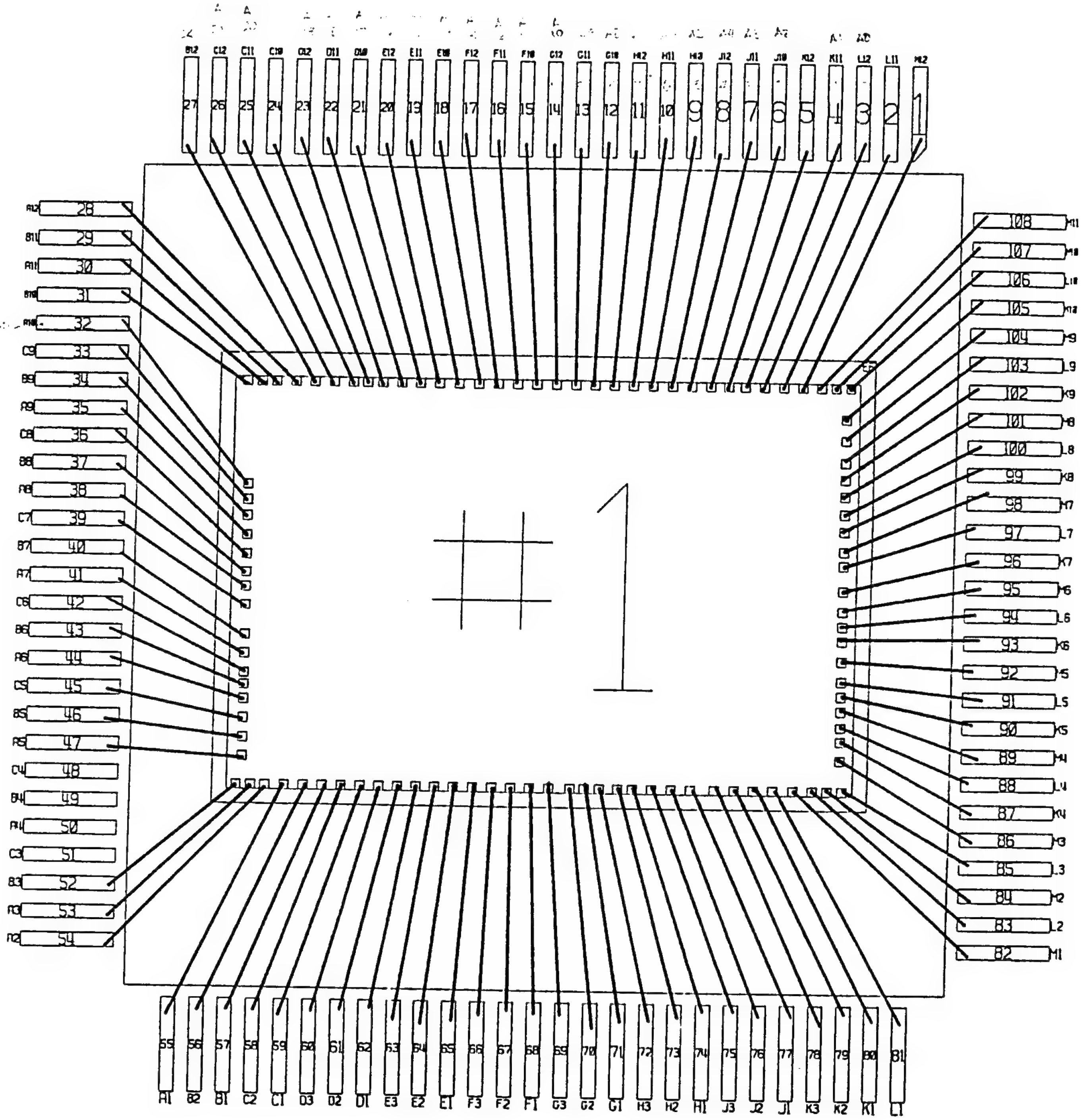
This appendix contains the Read Prediction Buffer Pinout by name and Description

Read Prediction Buffer Pinout by Name and Description (108 Pin PGA Package)

| PIN # | GRID # | NAME | PIN # | GRID # | NAME | PIN # | GRID # | NAME |
|---|---|---|---|---|---|---|---|---|
| 1 | M-12 | F1 | 37 | B-8 | Vdd | 73 | H-2 | GND |
| 2 | L-11 | F0 | 38 | A-8 | GND | 74 | H-1 | K6 |
| 3 | L-12 | A0 | 39 | C-7 | MACE | 75 | J-3 | K5 |
| 4 | K-11 | A1 | 40 | B-7 | DV | 76 | J-2 | K4 |
| 5 | K-12 | GND | 41 | A-7 | EWR | 77 | J-1 | K3 |
| 6 | J-10 | A2 | 42 | C-6 | Vdd | 78 | K-3 | K2 |
| 7 | J-11 | A3 | 43 | B-6 | GND | 79 | K-2 | K1 |
| 8 | J-12 | A4 | 44 | A-6 | EMA | 80 | K-1 | K0 |
| 9 | H-10 | A5 | 45 | C-5 | ROK | 81 | L-1 | F21 |
| 10 | H-11 | A6 | 46 | B-5 | NDV | 82 | M-1 | F20 |
| 11 | H-12 | A7 | 47 | A-5 | G8 | 83 | L-2 | F19 |
| 12 | G-10 | A8 | 48 | C-4 | N/C | 84 | M-2 | Vdd |
| 13 | G-11 | A9 | 49 | B-4 | N/C | 85 | L-3 | F18 |
| 14 | G-12 | A10 | 50 | A-4 | N/C | 86 | M-3 | F17 |
| 15 | F-10 | A11 | 51 | C-3 | N/c | 87 | K-4 | F16 |
| 16 | F-11 | A12 | 52 | B-3 | G7 | 88 | L-4 | GND |
| 17 | F-12 | A13 | 53 | A-3 | Vdd | 89 | M-4 | F15 |
| 18 | E-10 | A14 | 54 | A-2 | G6 | 90 | K-5 | Vdd |
| 19 | E-11 | A15 | 55 | A-1 | G5 | 91 | L-5 | F14 |
| 20 | E-12 | A16 | 56 | B-2 | G4 | 92 | M-5 | F13 |
| 21 | D-10 | A17 | 57 | B-1 | G3 | 93 | K-6 | F12 |
| 22 | D-11 | A18 | 58 | C-2 | G2 | 94 | L-6 | GND |
| 23 | D-12 | A19 | 59 | C-1 | G1 | 95 | M-6 | F11 |
| 24 | C-10 | GND | 60 | D-3 | G0 | 96 | K-7 | F10 |
| 25 | C-11 | A20 | 61 | D-2 | H8 | 97 | L-7 | GND |
| 26 | C-12 | A21 | 62 | D-1 | H7 | 98 | M-7 | F9 |
| 27 | B-12 | clkb | 63 | E-3 | H6 | 99 | K-8 | F8 |
| 28 | A-12 | NAV | 64 | E-2 | H5 | 100 | L-8 | Vdd |
| 29 | B-11 | MAC | 65 | E-1 | H4 | 101 | M-8 | F7 |
| 30 | A-11 | Vdd | 66 | F-3 | H3 | 102 | K-9 | Vdd |
| 31 | B-10 | NRF | 67 | F-2 | H2 | 103 | L-9 | F6 |
| 32 | A-10 | GND | 68 | F-1 | H1 | 104 | M-9 | F5 |
| 33 | C-9 | AV | 69 | G-3 | H0 | 105 | K-10 | F4 |
| 34 | B-9 | Vdd | 70 | G-2 | K8 | 106 | L-10 | F3 |
| 35 | A-9 | RDWR | 71 | G-1 | GND | 107 | M-10 | Vdd |
| 36 | C-8 | clka | 72 | H-3 | K7 | 108 | M-11 | F2 |

Functional Names:

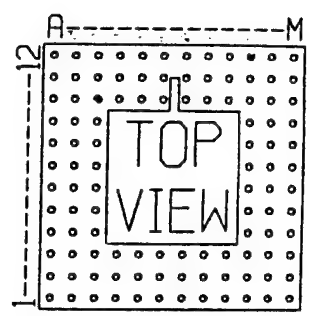
A21-A0: Address Bus from CPU
F21-F0: Output of RPB Address MUX to Main Memory
G8-G0: Data Bus from CPU
H8-H0: Data Read from Main Memory
K8-K0: Output of RPB Data MUX to Cache Memory
AV: Address Valid
NAV: Not Address Valid
DV: Data Valid
NDV: Not Data Valid
MAC: Memory Access Complete
NRF: No Refresh
RDWR: Not Read/Write
MACE: Memory Access Complete Early
EWR: Enable Write
EMA: Enable Memory Access
ROK: Refresh OK
clka: 110000
clkb: 000110
GND: Ground
Vdd: +5V




N340EE 2

309-NSF-CLASS/NPS

#1: 39343/YANG/CACHECONT




PGA108M: 12 PARTS

30-APR-1993

# APPENDIX E. EPOCH'S COMMANDS

This appendix contains all the commands necessary to access the Epoch's Floor-Planner in order to view the created geometries.

The following is an example of how to remotely access a Computer Center machine (sp254207) and execute Epoch:

ac6:/users/work3/aguilar>> **xhost sp254207.cc.nps.navy.mil**

sp254207.cc.nps.navy.mil being added to access control list

ac6:/users/work3/aguilar>> **rlogin sp254207.cc.nps.navy.mil**

login: **meaguila**

Password:******

---------------------------------------------------------------------

There are new Computer Center messages.

Type in ccmsgs to read the new messages.

<102 sp254207(SunOS) /kepler_u0/meaguila> **setenv DISPLAY ac6.cs:0.0**

<103 sp254207(SunOS) /kepler_u0/meaguila> **cd projects**

<104 sp254207(SunOS) /meaguila/projects> **cepoch**

Once Epoch's GUI is display, select from the pull down menu bar;

**Project-->project--> open**

a pop up window appears. Select (click on) from the "Existing Projects" selection window the following option:

**/tmp_mnt/h/kepler_u0/meaguila/projects/PRC2 c03**

click on "OK" push button and the window goes away. From the initial GUI window select;

**Physical Design-->Manual Compile--> Floor Planning...**

A new window should appear (it actually takes a few seconds to appear). From the new window's pull down menu bar select:

**Project Manager-->Parts--> Edit ...**

A pop up window appears. The created designs are listed in the "Existing Designs" selection window. Select desired one, then click OK. A splash window appears. Wait until the cell is loaded (it may take some time depending on the size of the design). The design appears as a block level. Use the **view-->Geo Viewing --> Expand** All option to expand all the cells. Use the rest of the "view" commands to Zoom-in, un-expand, etc. To exit select quit from the "Project Manager" option.

# LIST OF REFERENCES


1 G.J. Nowicki, "The Design and Implementation of a Read Prediction Buffer," M.S. Thesis, U.S. Naval Postgraduate School, Monterey, CA (December 1992).

2 D.J. Fouts and A. Billingsley, "Predictive Read Caches: An Alternative to On-Chip Second-Level Caches Memories," *Journal of Microelectronic Systems Integration*, Vol. 2, No. 2, December 1994.

3 HP16500B/16501A Logic Analysis System, *User's Reference Manual,* Hewlet Packard, Colorado Springs, CO, May 1993.

4 HP16550A 100-MHz State/500-MHz Timing Logic Analyzer, *User's Reference Manual,* Hewlet Packard, Colorado Springs, CO, May 1993.

5 HP16520A/HP16521A Patter Generator, *User's Reference Manual,* Hewlet Packard, Colorado Springs, CO, May 1993.

6 A. Cockcroft, *Sun Performance and Tuning: SPARC and Solaris,* Prentice-Hall, Englewood Cliffs, NJ, 1994.

7 DSP Architect, *User's Reference Manual,* Mentor Graphics Corporation, Beaverton, OR, 1993.

8 Verilog-XL, *Student Manual,* Cadence Design Systems, Inc., San Jose, CA, October 1990.

9 Epoch 3.1, *User's Manual,* Cascade Designs Automation, Bellevue, WA, 1994.

10 PowerPC-603, *User's Manual,* IBM Microelectronics and Motorola, 1994.

11 A. S. Tanenbaum, *Modern Operating Systems,* Prentice-Hall, Englewood Cliffs, NJ., pp. 110-111, 1992.

12 J. F. Wakerly, *Digital Design; Principles & Practices,* Second Edition, Prentice-Hall, Englewood Cliffs, NJ., pp. 320-323, 1994.

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library — 2
   Code 52
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Chairman, Code EC — 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

4. Chairman, Code CS — 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5118

5. Dr Douglas Fouts, Code EC/Fs — 3
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

6. Dr Timothy Shimeall, Code CS/Sm — 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5118